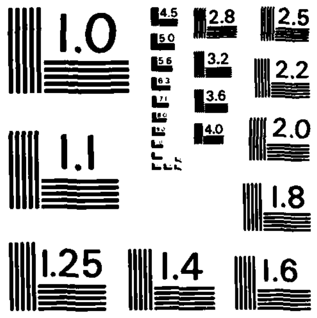
AD-A147 648 KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT VOLUME 4(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 30 APR 84 1/5

UNCLASSIFIED NOSC/TD-552-VOL-4 F/G 9/2 NL


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

NOSC TD 552

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT

VOLUME IV

Patricia Oberndorf, KIT Chairman
Naval Ocean Systems Center
San Diego, CA 92152

30 April 1984

Interim Report for 1 July 1983 — 30 April 1984

Prepared for
ADA JOINT PROGRAM OFFICE
3D139 (400AN) Pentagon
Washington, DC 20301

Approved for public release; distribution unlimited

NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN
Commander

R.M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This report is the fourth in a series consisting of inputs from the KAPSE Interface Team and its auxiliary industry/academia team. The work was sponsored by the Ada Joint Program Office under program element RDAF, project CS22, task AF0038AJPO, unit 84-05. The contributions are reproduced here exactly as received.

Released by	Under authority of
R. A. Wasilausky, Head	J. Stawiski, Head
C^3I Support Systems	Command, Control, Communications
Engineering Division	and Intelligence Systems Department

ACKNOWLEDGEMENTS

The author extends her appreciation to the many DOD, academic, and commercial activities whose continued support makes this effort possible. The sense of teamwork and cooperation displayed by all members of these two teams is outstanding and will mean the success of the program.

MM

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NOSC TD 552	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Ocean Systems Center; 6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): San Diego, CA 92152	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Ada Joint Program Office; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code): 3D139 (400AN) Pentagon, Washington, DC 20301	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
RDAF	CS22	AF0038AJPO	84-05

11. TITLE (Include Security Classification)
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT
Volume IV

12. PERSONAL AUTHOR(S)
Patricia Oberndorf, KIT Chairman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM 1 JUL 83 TO 30 APR 84	30 April 1984	436

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Computer language	KIT
Ada	KITIA
Interface standards	KAPSE
Programming support systems	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The continuing activities of the Kernel Ada Programming Support Environments (KAPSE) interface team and its industry/academia auxiliary are reported. ~~(Ada is a recent, DOD-developed programming language.)~~ The Ada Joint Program Office (AJPO)-sponsored effort will ensure the interoperability and transportability of tools and data bases among different KAPSE implementations. The effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous ADA-related developments, the widest possible participation is being encouraged to create a broad base of experience and acceptance in industry, academia, and the DOD. → Ce p 1-2

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED	Agency Accession
22a. NAME OF RESPONSIBLE INDIVIDUAL: Patricia Oberndorf	22b. TELEPHONE (Include Area Code): (619) 225-6682	22c. OFFICE SYMBOL

DD FORM 1473, 84 JAN   83 APR EDITION MAY BE USED UNTIL EXHAUSTED. ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DD FORM 1473, 84 JAN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Technical Report

APSE I & T
TECHNICAL PROGRESS

30 April 1984

Contract # N66001-83-C-0245
CDRL Item A005

for

NAVAL OCEAN SYSTEMS CENTER
271 Catalina Boulevard
San Diego, California 92152

A-1

Prepared by

TRW
Systems Engineering & Applications Division
Defense Systems Group
3420 Kenyon Street, Suite 202
San Diego, California 92110

CONTENTS

CONTENTS
(Concluded)

SECTION 1

INTRODUCTION

INTRODUCTION

This report is the fourth in a series that is being published by the KAPSE Interface Team (KIT). The first was published as a Naval Ocean Systems Center (NOSC) Report, TD-209, dated April 1982, and is now available through the National Technical Information Service (NTIS)* for $19.50 hardcopy or $4.00 microfiche; ask for order number AD A115 590. The second was published as NOSC TD-552, dated October 1982, and is now available through NTIS for $44.50 hardcopy; ask for order number AD A123 136. The third was also published as NOSC TD-552, dated October 1983, and will be available through NTIS. This series of reports serves to record the activities which have taken place to date and to submit for public review the products that have resulted. The reports are issued approximately every six months. They should be viewed as snapshots of the progress of the KIT and its companion team, the KAPSE Interface Team from Industry and Academia (KITIA); everything that is ready for public review at a given time is included. These reports represent evolving ideas, so the contents should not be taken as fixed or final.

MEETINGS

The two teams are now regularly meeting jointly. Meetings during this reporting period include ones in July 1983 in San Diego, in October 1983 in Dallas, in January 1984 in San Diego and in April 1984 in Seattle. The approved minutes from the July, October and January meetings are included in this report. As usual, some of the working groups have also held individual meetings between regular KIT/KITIA meetings.

TEAM ORGANIZATION

During the July 1983 meeting, a new set of joint KIT/KITIA working groups was organized. Following the lead of the joint KIT/KITIA working group which had taken responsibility for the Common APSE Interface Set (CAIS - formerly SIS), several other working groups were formed to take responsibility in other product or work areas. Throughout much of the rest of this report, you will find references to:

*National Technical Information Service (NTIS), Department of Commerce, 5285 Port Royal Road, Springfield, Va. 22161

CAISWG:	the CAIS working group, responsible for producing various versions of the CAIS
RACWG:	the Requirements and Design Criteria working group, responsible for the production of the requirements document which will guide the development of version 2 of the CAIS
GACWG:	the Guidelines and Conventions working group, working to produce a User's Guide to Ada Transportability
DEFWG:	the Definitions working group, responsible for bringing together a glossary of terms found in the KIT/KITIA documents so that the terminology is used consistently and accurately
STONEWG:	the STONEMAN working group, responsible for reviewing STONEMAN with the requirements of I&T in mind and suggesting improvements which provide a better context for the work of the KIT and KITIA
COMPWG:	the Compliance working group, studying the implications of trying to validate the conformance of a particular CAIS implementation to the CAIS standard
STANDWG:	the Standards working group, responsible for guiding the teams with respect to proper procedures and formats for standardization of the CAIS specification as well as making sure the teams are aware of existing standards which are closely related to the CAIS

THE APSE INTEROPERABILITY AND TRANSPORTABILITY (I&T) PLAN

A new version of this plan is included in this report. It reflects only minor changes in thinking, as well as the new schedule given to the teams by the Tri-Service Review board in November 1983.

THE COMMON APSE INTERFACE SET (CAIS)

Formerly the Standard Interface Set (SIS), the name has been changed by which the interface set is known. It was felt that putting the word "standard" in the title before the interface set was declared a standard was misleading. The new term is pronounced "CASE".

The CAISWG met several times during 1983 in order to finalize an initial version of the CAIS. This version was documented as CAIS Version 1.Ø and was presented at a public review meeting in Washington D.C. in September 1983. This document contained several errata which were corrected in CAIS Version 1.1. This version is included in this report and can also be ordered from NTIS using order number A134 825.

This public review was well-attended, and several comments and questions were taken by the CAISWG members. Probably the most consistent criticisms levelled at CAIS 1.1 are (1) that the semantics are incomplete and (2) that a rationale should be provided which would provide the sort of explanation which the presenters gave during the review presentations. Both of these aspects are being worked on by the CAISWG, and future versions are expected to show improvements. Comments received on the CAIS document can be found through the MILNET account KIT-INFORMATION, password "KIT".

Subsequent to the public review, the Tri-Service Review board met to consider the schedules of various activities in relation to the proposed CAIS schedule. As a result a new CAIS schedule was given to the KIT and KITIA:

Draft CAIS 1.2	-	April 1984
Draft CAIS 1.3	-	November 1984
MIL-STD CAIS 1	-	January 1985
Start CAIS 2.Ø	-	January 1985
Draft CAIS 2.Ø	-	January 1986
MIL-STD CAIS 2	-	January 1987

These changes were prompted primarily by a desire to better coordinate the CAIS production with contractual obligations for the ALS, ALS/N and AIE. The new schedule is based on the production of two MIL-STDs, the first after three years of KIT/KITIA work and the other after 2 more years; the original schedule called for one MIL-STD after four years of work. The November meeting results are included in this report as a Memorandum for the Record from Bob Mathis, AJPO Director.

In response to these changes, the CAISWG has somewhat loosened the restrictions to strict compatibility with the AIE and ALS for version 1. This is necessitated by two main facts:

- if version 1 is to be standardized, it must be sufficiently complete to (1) stand up on its own and (2) go far enough to make upward compatibility with version 2 a possibility

- this makes it necessary for version 1 to venture into some areas in which there is not good agreement between the AIE and ALS, resulting in departure from one or the other or possibly both in an attempt to find an acceptable compromise.

REQUIREMENTS AND CRITERIA (RAC)

Further clarification of issues and consensus have emerged from the last several months of debate on the RAC. Another attempt at a new organization was made in December 1983, modelling the RAC on the STEELMAN document which gave the final requirements for the Ada language. This organization has proven quite successful and has lead to the ability of the KIT and KITIA members to concentrate on the issues at hand more clearly than ever before. However, although several parts of the document have matured sufficiently for a vote to be taken in April 1984, a lot more work remains to be done.

The version of the RAC which was voted on in April appears in this report. It is accompanied by an introduction which explains some of what transpired in the course of the April meeting concerning the RAC. The reader will notice that two sections were not considered mature enough in April to warrant a KIT/KITIA vote. Of these, the most difficult to deal with is the one on the KAPSE database (i.e., the section entitled Object Management). These requirements have been the most elusive and frustrating to the teams. The latest (all new) version was only introduced at the January meeting, but it is starting to gather consensus. It is most likely that this is the most difficult area because no exact model for such a thing exists. It shows some of the characteristics of both a classical operating system and a modern data base management system, but cannot be treated as either one. It is hoped that an appropriate model will be articulated as a result of the work done at the April meeting.

STONEMAN

One of the problems the teams have faced repeatedly is the question of the relationship between STONEMAN and the KIT/KITIA charter. While it is clear that STONEMAN set the stage for the entire APSE effort - and therefore of the I&T work - it is equally clear that STONEMAN did not deal with a number of issues which have emerged now, four years later. Among these are the existence of at least two KAPSE-incompatible APSEs, the trend towards workstations and the distribution of APSEs over a variety of machines, and the DoD's emerging requirements concerning discretionary and mandatory security in development environments.

The need to more closely examine STONEMAN was particularly evident as the teams worked to develop the RAC. Periodically, the RACWG has come upon requirements which are needed to achieve I&T but are more general in scope. It was natural to look to STONEMAN to provide such requirements, but it was often found that STONEMAN was mute on the point. This lead to the formation of the STONEWG to investigate these and other aspects of the 1980 document in an attempt to determine what improvements could be made after four years of experience, including the current I&T effort. The initial STONEWG analysis of STONEMAN appears in this report, along with a report on the status and plans of the STONEWG.

GUIDELINES AND CONVENTIONS

Very early in the teams' work on I&T it was realized that there was a limit to the I&T which can be achieved simply by declaring a set of interface standards. It has been the intention of the teams to publish such "wisdom" and guidelines as they have accumulated in the course of working on the I&T problem, in order to help those interested in achieving I&T beyond use of the interfaces. The GACWG was formed to undertake the development of such a set of guidelines, and the document will be called the User's Guide to Ada Transportability. A proposed outline for this document appears in this report.

COMPLIANCE

In the tradition of the Ada language, it has been a goal of this effort to produce a set of interface specifications against which an implementation could be judged to determine its conformance. The focus of this effort, as shown in the COMPWG's charter in this report, is to gu`de the production of the CAIS documents in such a way that the resulting product can be used to validate implementations. Special emphasis is placed on completeness and consistency of the document, particularly the semantics. Because the area of semantics is such a difficult one, there have been several KIT/KITIA papers concerning it in previous public reports. Two more, by Freedman and Yelowitz, are included in this report.

DEFINITIONS

One of the first problems with which the teams dealt was that of definitions. This resulted in the inclusion of definitions for such terms as Transportability and Interoperability in an earlier public report. This work is continuing, under the guidance of the DEFWG. The goal is to produce a large pool of terms used throughout the products of the KIT/KITIA, with a glossary of the terms used in a particular document included with that document. The first version of such a pool of terms is included in this report. Wherever possible, the terms have been taken from other existing glossaries, but some are used in a unique manner in KIT/KITIA documents and are so defined in the pool.

STANDARDS

It is important that the work of the teams be coordinated with other work in the area of standards. The STANDWG serves as the liaison between the KIT/KITIA and the standards community. The working group is collecting a list of possibly related standards, as well as studying the existing military documents on how one creates and formats a standard. This guidance will be factored into the work of the CAISWG.

POLICY DISCUSSIONS

It seems that policy is never far from anyone's mind when discussing the Ada program, and this has lead to many discussions during the KIT and KITIA meetings. Such concerns lead to the generation of an I&T Strategy document (published in a previous public report). In October 1983, the announcements by the three services of their intended GFE policies for the APSEs lead to considerable concern among industry representatives. This, coupled with the DoD report to Congress in December 1983, resulted in a new KITIA policy paper by Wrege and a new KITIA policy recommendation to the AJPO. These are included in this report.

KIT AND KITIA PAPERS

In addition to the two semantics papers already mentioned, several other papers have been produced by members of the KIT and KITIA. Two concern security questions and how to impose security requirements on the CAIS. These are by Pasterchick and Fitch. Another, by Gargaro, is a suggested re-ordering for the RAC which was used in that document's evolution. Following the semantics papers, the reader will find a paper which compares the CAIS to UNIX and raises some questions about how one might go about using the UNIX interfaces as a more obvious basis for CAIS version 1.

I&T TOOLS

The work by CSC in the area of Configuration Management has been concluded. The final report on the Configuration Management Workshop held in June 1983 is included in this report.

The work by TI on the APSE Interactive Monitor (AIM) is continuing. Due to the difficulties of securing an appropriate compilation capability, the actual implementation of the AIM has been delayed, but plans are now taking shape which should see an AIM implementation some time in early 1985.

OTHER KIT/KITIA ACTIVITIES

Two other activities were spurred by the DoD report to Congress mentioned under Policy Discussions above. One was an attempt to "open" the KITIA up to wider participation and involvement; this was characterized as a KITIA extension, nick-named "KITIA-X". The other was an attempt to start planning for an Ada Run-Time Environment Working Group (ARTEWG); this is specifically called out in the report to Congress as one of several panels to be included in a new DoD Computer Systems Interface Working Group (CSIWG). While proposals for the

KITIA extension were formally rejected at the April 1984 KITIA meeting (a more informal approach was approved), work did begin to plan for the ARTEWG. The proposed ARTEWG charter is included here.

Another aspect of APSE investigation which has been pursued in conjunction with the KIT is the periodic review of both the AIE and the ALS. Included here is a report on the latest of these, a review of the new AIE B-5 specifications issued following the contract award.

Finally, in response to many inquiries, the AJPO has decided to foster the organization of a CAIS Implementors' Group. Those who have spoken to either the AJPO or NOSC about interest in being part of such a group have been invited to an initial organizational meeting on 18 June 1984 in San Diego. This one-day meeting will begin with introductions of what each person is doing concerning the CAIS, but the major emphasis will be on organizing a group which will then take responsibility for itself; it is not intended that this group will in any way be a part of the KIT or KITIA, although there is great KIT/KITIA interest in and desire for interaction with those who are learning about the CAIS through implementation.

CONCLUSION

This Public Report is provided by the KIT and KITIA to solicit comments and feedback from those who do not regularly participate on either of the teams. Comments on this and all previous reports are encouraged. They should be addressed to:

Patricia Oberndorf
Code 423
Naval Ocean Systems Center
San Diego, CA 92152

or sent via ARPANET/MILNET to POBERNDORF@ECLB.

SECTION 2

TEAM PROCEEDINGS

KIT/KITIA MINUTES
MEETING OF 12-14 JULY 1983
SAN DIEGO, CALIFORNIA

ATTENDEES: SEE APPENDIX A
BIBLIOGRAPHY OF HANDOUTS: SEE APPENDIX B

12 JULY 1983

1. OPENING REMARKS

Tricia Oberndorf, KIT Chairperson, brought the meeting to order.

The KAPSE Interface Team Public Report will be available shortly.

The results of the AIE Public Review are being compiled by Warren Loper.

Additional HERMES documents have been ordered.

The KIT/KITIA will form new "named" working groups to work in parallel with the present numbered working groups. As well as the existing SISWG (for definition of the initial Standard Interface Set), there will be new working groups to address Requirements and Criteria (RACWG), STONEMAN II (STONEWG), Definitions (DEFWG), Standards (STANDWG), and Compliance (COMPWG).

Brian Schaar, AJPO, made the following announcements:

- The U.S. Army is making the ALS available to various companies that agree to certain conditions on its use as contained in an announcement in the Commerce Business Daily.
- Bob Mathis, AJPO, has determined there will be a Public Review of the Standard Interface Set document in Washington on the 14th & 15th of September 1983. Therefore the SISWG will have a dry run in Boston in late August to prepare for this review. The first day will be dedicated to presenting the SIS model followed by panel discussions on the second day.
- The AJPO is now forming an APSE Evaluation and Validation Team similar to the KIT. The group is presently Air Force led.
- Lt. Col. Vance Mall is the new Deputy Director of STARS.

2. GENERAL BUSINESS

In reading the SIS document, KIT/KITIA members are reminded of the basic guidelines the SISWG had to follow: implementable on the AIE/ALS, a bare machine, and modern operating systems.

The SIS is available on the KIT-INFORMATION directory.

A review of the current version of the Interoperability & Transportability Management Plan was conducted.

The results of the previous day's RACWG meeting were presented. All comments on the R&C document should go to numbered WG chairs.

The SISWG will present their latest decisions after lunch. If there are specific questions in a particular area, the appropriate SISWG member should be contacted at lunch.

3. BREAK FOR LUNCH

4. GENERAL DISCUSSIONS ON THE SIS AND R&C DOCUMENTS

Working Group 3 suggested the SIS packages be organized into groups so that the document would allow a building of libraries of packages. The R&C document should have more functionally based guidelines.

Anthony Gargaro presented areas for the SIS to improve in its consistency with the Ada RM including consistent use of parameter modes, range constants, and type declarations.

A discussion of the different sections of the SIS followed, including the definition of the SIS before the R&C document and the use of Ada RM Text IO packages. A distinction between the initial SIS and the SIS to which the R&C document applies was reiterated.

If there are significant disagreements between the work of the SIS and opinions of KIT/KITIA members, there is still the "minority report" vehicle available to all KIT/KITIA members to document their concern.

Tricia Oberndorf presented the results of the KIT/KITIA canvass she conducted to insure commonality of goals.

5. WORKING GROUP REPORTS

Working Group 1 concentrated on the Process Management, TEXT IO, and device IO sections. Tim Harrison discussed the rationale for inclusion of specific procedures in response to WG1 comments.

Working Group 2 requested the Process Model be discussed further before specific comments were submitted.

Working Group 3 presented discussion of the inter-process communications area which was further defined by the SISWG.

Working Group 4 expressed a desire for additional rationale and semantics throughout the document. Extensibility issues have not been addressed in this version of the SIS. Facilities are missing for a broadcast message, asynchronous communications, security, and recovery mechanisms. SISWG admitted these items were considered but deferred in this version.

Working Group 2 voiced serious reservations concerning the feasibility of the database model. Major points raised were binary relations, attributes on relations, typing on attributes, binding of names to attributes and relationships. Other points (i.e., runtime typing, access control, and different stategies of primary relation concept) were not greatly discussed by Working Group 2.

--Erhard Ploedereder commented on Working Group 2's comment of

"Serious reservations". He believes that the database model is one we can live with; it certainly deserves evolution, but the entire approach is NOT gravely faulty.

A presentation by Tucker Taft followed on Name Spaces. He pointed out the reasons to bring process and data base models together: to keep process and data naming similar, to keep relationships between processes and between processes and data bases similar, and to cut down on the number of distinct packages. He also discussed syntax of the name space.

Dit Morse presented an Execution Model produced by Bill Wilder which dealt with mechanisms for process execution and mechanisms for an Ada-like task execution.

Friday

6. GENERAL BUSINESS

Presentation on the Standardization Process by Bill LaPlant were given.

7. BREAK FOR MORNING WORKING GROUP MEETINGS

8. LUNCH

9. WORKING GROUP REPORTS

SISWG presented an outline of the new Sections of the SIS and declined from mentioning further availability of the document until further discussion has taken place among the group members.

RACWG announced their editor, Reed Kotler. The RACWG plans to narrow systems requirements. They stress the SIS is an interface, NOT an implementation, and in addition to I & T requirements there should be fuctionality requirements.

STONEWG's purpose is to produce and update a document stating the requirements for Ada Programming Support Environments. They plan to use Stoneman 1980 Version (the original Stoneman) and other available APSE documents. Suggested topics are a definitions of terms, Ada program environments, and host environments.

DEFWG expressed four points. One, the problem- a need for definitions (good light vs. chandelier). Two, a purpose- a clear consistent glossary of critical terms across and within KIT/KITIA documents and to oversee the use of this glossary. Three, establish mechanisms for determining a necessary glossary of terms. Four, the glossary should be printed as a feature of the Public Report and terms in the Public Report should be reserved. Suggestions were also made for a terminologist from each working group.

STANDWG, consisting of 2 members, elaborated on this morning's presentation of the standardization process mentioning the formation of a KIT/KITIA working group on Standardization. They discussed the need to develop a facility for reformatting a document for standardization and to use current standards for this reformation.

Guidelines Working Group will update a current outline written by Ron Johnson of the Guidelines document and make it available at the October meeting.

COMPWG discussed evaluations and validations of APSEs and SISs and the process of defining requirements of a validation . Points expressed were to develop a compliance matrix, and to develop a methodology for the specification of KAPSE interface semantics so they can also be used with the R & C.

10. MEETING WAS ADJOURNED AND WORKING GROUP MEETINGS FOLLOWED.

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
12-14 July 1983

KIT Attendees:

CASTOR, Jinny	AFWAL/AAAF
FERGUSON, Jay	DoD
FOIDL, Jack	TRW
FOREMAN, John	Texas Instruments
FROMHOLD, Barbara	U.S. Army CECOM
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
JOHNSON, Doug	SoftWrights
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LAPLANT, Bill	HQ USAF
LOPER, Warren	NOSC
MAGLIERI, Lucas	Canadian National Defense HQ
MILLER, Jo	NWC
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
ROBERTSON, George	FCDSSA-SD
SCHAAR, Brian	AJPO
STEIN, Mo	NSWC/DL
STOPYRA, Norma	NAVMAT (Guest)
TAFT, Tucker	Intermetrics

TAYLOR, Guy	FCDSSA-DN
THALL, Rich	SofTech
WILDER, Bill	SofTech

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnel Douglas Astronautics
BEANE, John	Honeywell
COX, Fred	Georgia Institute of Technology
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Sytsems
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	Computer Sciences Corp.
HUMPHREY, Dianna	Control Data Corp.
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles & Space
LAMB, J. Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
LYONS, Tim	Software Sciences Ltd., U.K.
McGONAGLE, Dave	General Electric
MORSE, H. R.	Frey Federal Systems
PLOEDEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
RUBY, Jim	Hughes Aircraft Co.
SAIB, Sabina	General Research Corp.
SIBLEY, Edgar	Alpha Omega Group, Inc.
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

APPENDIX B - MEETING HANDOUTS

1. SIS Drafters Working Papers, SIS Working Group
2. "Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for the Standard Interface Set", WORKING PAPER, dtd. 27 June 1983
3. "Computer Programming Language Policy", DoD Directive 5000.31, Draft Revision
4. "SIS Categories", Preliminary Draft, D.E. Wrege
5. Hardcopy of presentation "Document Control Methodology" by J. Foidl

KIT/KITIA MINUTES
MEETING OF 16-18 OCTOBER 1983
DALLAS, TEXAS

ATTENDEES: SEE APPENDIX A
HANDOUTS: SEE APPENDIX B

16 OCTOBER 1983

1. OPENING REMARKS

Tricia Oberndorf, KIT Chairperson, brought the meeting to order.

John Foreman, Texas Instruments, was introduced as the local host for the KIT/KITIA and Ada-TEC meetings.

2. GENERAL BUSINESS

The Computer Sciences Corporation contract for Ada tool development has been completed through the design stage. A Configuration Management workshop was held and a report is to be generated on its results.

Texas Instruments has progressed through the Critical Design Review in the development of the Ada Interactive Monitor tool.

The Request for Proposal for the expected third Ada tool has been delayed indefinitely.

The Ada Language System (ALS) is due to be delivered to the Army this month. Copies of the ALS are available from the Army under the terms described in the Commerce Business Daily.

3. WORKING GROUP REPORTS

CAISWG - Common APSE Interface Set Working Group - conducted a Public Review of the Common APSE Interface Set (CAIS) Version 1 in Washington, D.C. in mid-September. The review attracted 217 attendees from government and industry who submitted 125 questions which were answered at the Review. The final revision to the Version 1 document is expected to be released in April 1984. CAIS Version 2 work is expected to commence in January 1984 for ultimate submission as a DoD Standard. Review comments on CAIS Version 1 are requested from industry by 1 November 1983 and from DoD by 15 December 1983. Implementation of Version 1 is at the user's own risk.

RACWG - Requirements and Criteria Working Group - circulated a point paper from Working Group 1 (A. Gargaro) on "CAIS Program Transportability Interface" and requested all KIT/KITIA members to contribute to the revision of the R&C document presently under way. The target date for submission of this document to the KIT/KITIA for final approval is presently January 1984.

DEFWG - Definition Working Group - has a list of terms available for review and has obtained a good glossary from the U.K. Study Report. DEFWG will provide hard copies to the working group chairs.

COMPWG - Compliance Working Group - is in the formulation stage with a meeting scheduled for Washington in mid-December.

STANDWG - Standards Working Group - distributed hardcopies of the slide presentation on the "Defense Standardization Program" presented at the last KIT/KITIA meeting in San Diego, and a trip report including standardization recommmendations to the KIT and KITIA.

GACWG - Guideline and Conventions Working Group - report deferred.

Working Group 1 - reviewing inputs for the R&C document.

Working Group 2 (KIT) - in a transition phase.
(KITIA) - no report.

Working Group 3 (KIT) - in a transition phase.
(KITIA) - reviewing named working group products.

Working Group 4 (KIT) - no report.
(KITIA) - no report.

4. AJPO STATUS

Members may experience some problems with the split of the ARPANET into the MILNET and ARPANET. Best to work with your local TAC representative whose names are listed in the ARPANET handbook or via NIC@NIC.

Brian Schaar (AJPO) expressed his gratitude to John Foreman (TI) for his efforts in hosting the KIT/KITIA Meeting and the AdaTEC. He also expressed his thanks for the efforts of the CAISWG in the generation of the Version 1 document. The feedback on th CAIS Public Review has been positive thus far.

Dr. Edith Martin (USDRE) will be in Dallas to present DoD Certificates of Recognition to individuals who contributed to the Ada language definition.

Dr. Robert Mathis (AJPO) will be attending some of tomorrow's working group meetings.

A tri-service meeting is planned for Thursday at which the CAIS will be formally submitted to the services for review.

There will be a CAIS presentation to the AdaTEC attendees on Thursday morning.

5. BREAK FOR DINNER AND WORKING GROUP MEETINGS

MONDAY OCTOBER 17, 1983

6. CONTINUE WORKING GROUP MEETINGS

7. BREAK FOR LUNCH

8. SEPARATE KIT AND KITIA MEETINGS

9. CAIS PRESENTATIONS

Dr. R. Mathis, Director, Ada Joint Program Office, presented the members of the CAIS Working Group with covers of CAIS Version 1. The covers contained an expression of gratitude for the work on the document and were signed by Dr. Mathis (AJPO) and Dr. Martin (USDRE).

10. EUROPEAN ECONOMIC COMMITTEE PRESENTATION

The status of the Ada work being performed in Europe was presented by a member of the EEC which includes an Ada compiler and a portable MAPSE which is separate from the U.K. funded MAPSE efforts. Ada-Europe also supports such groups as AdaTEC in the ALS and AIE review cycles and will support the review of the CAIS version 1.

11. CAIS DISCUSSION

A freeform discussion on the status and plans for the future CAIS was held. Major areas of concern addressed the following topics. The final iteration of Version 1 cannot preclude future implementations of distributed environments although they are not specifically addressed in the document. The process control and communication areas need additional work. The concept of the "node handle" needs additional clarification. The generic device node mechanism needs completion. The area of discretionary access control needs identification, and mandatory access control requires consideration. The need for network interfaces should be considered. Standard attributes should be defined as part of the document (and possible standard exceptions and relation names). Resource access and resource synchronization should be considered for the final version.

12. BREAK FOR COFFEE

13. FUTURE MEETING SCHEDULE

Tricia Oberndorf defined the future KIT/KITA meeting schedule as follows:

16-19 Jan 84	San Diego
9-12 Apr 84	Seattle
16-19 Jul 84	Toronto
1-4 Oct 84	New Hampshire
Jan 85	San Diego
Apr 85	Washington, D.C.
Jul 85	San Francisco
Sep 85	Connecticut
Jan 86	San Diego

14. KIT/KITIA CAIS COMMENTS

A poll of the individual KIT/KITIA members was taken to identify the major concerns they held regarding the CAIS Version 1 document. The following is a summary of these potential problem areas:

- lack of security related functions in accordance with DoD security requirements
- how projects transition to an APSE implementing the CAIS
- the exact relationship of the contents of the CAIS to STONEMAN
- the apparent weakness of the inter-process model
- the impact on existing tools re: portability

- how to address interfaces not fundamentally addressed in the present CAIS
- lack of data management
- lack of a rationale format such as
  - issues
  - alternatives
  - advantages
  - disadvantages
- definition of policy regarding standardization and conformance
- need for graphics and data dictionary support
- need to address configuration management
- need to address distributed environments

TUESDAY, 18 OCTOBER

15. REQUIREMENTS AND CRITERIA DOCUMENT

A discussion of the Requirements and Criteria document was held to plan for the future revision of this document. The major concerns regarding the present document were the lack of pointers or cross-references to the STONEMAN document, reaffirmation of "criteria" as a basis of selecting between alternate CAIS implementations, and a reaffirmation of the definition of "interoperability" as used in the document.

16. RELATED R&C PRESENTATIONS

The following brief presentations and discussions were led by various RACWG members. Their purpose was to solicit feedback from the joint KIT/KITIA on the unresolved issues which the RACWG must resolve before the document can be finalized.

Anthony Gargaro (Computer Sciences Corporation) presented the methodology for the reorganization of the R&C document and a strategy for specification of requirements.

Dennis Cornhill (Honeywell) proposed a set of interfaces that may be required in the CAIS such as a standard editor, command language interpreter, and transmit/receive.

LCDR Philip Myers (Naval Electronics Systems Command) presented security concerns for APSEs including multi-level and non-multi-level security requirements. These requirements need inclusion in STONEMAN and the CAIS documents.

Edgar Sibley (Alpha Omega Group) presented a methodology for inclusion of a data management schema that can be extended for future versions of the CAIS.

17. BREAK

18. KIT/KITIA POLL

A question was put to the KIT and KITIA "Should there be requirements for the CAIS that would be the same for both the host and target run-time systems?". YEA - 15 NAY - 20

Regarding the definition of "Interoperability", should the KIT and KITIA use the definition as contained in the Public Report Volume 1, or as contained in CAIS Version 1? PR Vol 1 - 6 CAIS 1 - 14

Should the R&C document be defining requirements for CAIS Version 1, CAIS version 2, some future milestone (year), some future CAIS (Version X, where X>2)?

CAIS 1 - 0 CAIS 2 - 28 Year X - 0 CAIS X - 10

19. ADJOURN JOINT MEETING AND BREAK FOR WORKING GROUP MEETINGS

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
16-18 October 1983

KIT Attendees:

CASTOR, Jinny	AFWAL/AAAF
FERGUSON, Jay	DoD
FOIDL, Jack	TRW
FOREMAN, John	Texas Instruments
FROMHOLD, Barbara	U.S. Army CECOM
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
JOHNSON, Doug	SoftWrights
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LAPLANT, Bill	HQ USAF
LINDLEY, Larry	NAC
LOPER, Warren	NOSC
MAGLIERI, Lucas	Canadian National Defense HQ
MILLER, Jo	NWC
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PASTERCHIK, David	MITRE
PEELE, Shirley	FCDSSA-DN
ROBERTSON, George	FCDSSA-SD
SCHAAR, Brian	AJPO

STEIN, Mo	NSWC/DL
TAFT, Tucker	Intermetrics
TAYLOR, Guy	FCDSSA-DN
THALL, Rich	SofTech
WALTRIP, Chuck	John Hopkins Univ.
WILDER, Bill	SofTech

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas Astronautics
BEANE, John	Honeywell
CORNHILL, Dennis	Honeywell/SRC
COX, Fred	Georgia Institute of Technology
COCKERHAN, Beth	Georgia Institute of Technology
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Sytsems
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	Computer Sciences Corp.
GEHARDT, Mark	Raytheon
GOVELITY, Lori	IBM
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles & Space
LAHTINEN, Pekka	Oy Softplan AB Finland
LAMB, J. Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
MORSE, H. R.	Frey Federal Systems
PLEODEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
RUBY, Jim	Hughes Aircraft Co.
SAIB, Sabina	General Research Corp.
SAVAYA, John	Teledyne Systems
SIBLEY, Edgar	Alpha Omega Group, Inc.

WESTERMANN, Rob	TNO-IBBC The Netherlands
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

APPENDIX B - MEETING HANDOUTS

1. Common APSE Interface Set, Version 1.1, dtd. 30 September 1983

2. Hardcopy of presentation "Defense Standardization Program", W. LaPlant

3. "Security Requirements for Ada Programming Support Environments" DRAFT, David C. Pasterchik, dtd. 15 October 1983

4. "Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for the Common APSE Interface Set" WORKING PAPER, dtd. Columbus Day 1983

5. "CAIS Program Transportability Interface", KITIA Working Group 1 Position Paper, dtd. 30 August 1983

KIT/KITIA MINUTES
16-19 JANUARY 1984
SAN DIEGO, CALIFORNIA

ATTENDEES: SEE APPENDIX A
HANDOUTS: SEE APPENDIX B

16 JANUARY 1984 - KITIA MEETING

1. OPENING REMARKS

- Edgar Sibley, KITIA chairperson, brought the meeting to order. New members Steve Glaseman representing Aerospace Corporation and Andy Rudmik representing GTE (Phoenix) were welcomed and introduced. Skip Montanaro was introduced as the new alternate for Honeywell.

2. GENERAL BUSINESS

- Edgar reviewed the member selection process for the KITIA. The two new members were replacements for one release and one resignation. There were no objections to a KITIA size of thirty full members. Tim Lyons related that at the Ada Europe meeting it was proposed a member be added to the KITIA as an EEC Ada Europe representative. Edgar proposed the normal application process of resume submission to the KIT chairperson, management committee screening and a vote be followed. There were no objections to this proposal.

3. DoD POLICY DISCUSSIONS

- Eli Lamb said it is in DoD's best interest to take a non-exclusionary approach to environment acquisition for two critical reasons:

  1. to allow industry room for profit to motivate innovation and don't exclude competition

  2. to evolve the technology and encourage innovation

  The following recommendations were made:

  1. standardize Common APSE Interface Set (CAIS) as low-level (OS) interface and then evolve
  2. encourage development of toolsets based on CAIS
  3. require delivery of toolsets and software using CAIS
  4. GFE "demonstrably superior" tools
  5. make a clear statement that DoD is looking for "orders of magnitude" better environments

- Doug Wrege raised serious concerns with the DoD statements on GFE of environments. His key item was:

  1. a developer should develop software on any APSE as long as it can be maintained by the life-cycle APSE

- Doug recommended a draft policy statement from the KITIA be prepared prior to the January Tri-Service Review.

4. LAYERING FOR TOOLS

- Sabina Saib gave a presentation on Layering for Tools. This would allow for data abstraction at the lowest level and encourage building on the work of others. The major services required would be interaction, communications, and data management. Some of the needs of a tool developer such as character and string input, file opening, buffered IO, uniform tape format, uniform file naming and consistent file name parsing were also discussed.

5. BREAK FOR LUNCH

6. WORKING GROUP REPORTS

- Working Group 1 met in November in Washington. They have been reviewing the Requirements and Crtieria document from the perspective of functionality required for Interoperability and Transportability. They have reviewed CAIS Version 1.1. WG 1 believes they should continue as a reviewing body for KIT/KITIA products. The new WG1 chairman is Roy Freedman.

- Working Group 2 discussed the paper Edgar Sibley distributed on the NET. Judy Kerner was re-elected as the WG2 chairman.

- Working Group 3 announced Dick Drake as their new chair. WG 2 wants to maintain their review role on KIT/KITIA products.

- Working Group 4 is experiencing an identity crisis and may want to disband as a formal group. Nick Baker was announced as the new chairman.

- A discussion of the roles of the Working Groups followed including the roles of the "named" working groups (CAISWG, RACWG, etc.). It was recommended that the Working Groups meet tomorrow to decide their future organization and activities.

7. KITIA ELECTIONS

- Herm Fischer was elected as the new KITIA chairperson. The KITIA expressed their gratitude to Edgar Sibley for his excellent work as the former chair.

8. RULES DISCUSSION

- It was recommended the KITIA rules be clarified to define a "member" (an individual or an organization); clarify the voting role of the KIT chair and secretary; number of meetings missed; alternate attendance at meetings (one per year); clarify timeframes for secretary to publish data as at least two weeks before a meeting, minutes available three weeks after a meeting, send documentation to members missing a meeting, and provide a list of current documents.

9. KITIA POLICY STATEMENT

- Herm Fischer indicated a KITIA team would formulate a draft policy statement on DoD's intended GFE of environments for the KITIA to review later in the week.

10. KITIA MEETING ADJOURNED

17 JANUARY 1984 - JOINT KIT/KITIA MEETING

1. OPENING REMARKS

- Tricia Oberndorf, KIT chairperson, brought the meeting to order.

- Herm Fischer was introduced as the new KITIA chairperson.

- New members for the KITIA were introduced including A. Rudmik for GTE, S. Glaseman for Aerospace. New KIT members were also introduced including W. Wilder for PMS-408, Dennis Turner and John Hollister for CECOM, E. Lee as an alternate for NSWC, K. Chadwick as an alternate for Canadian National Defense HQ and F. Belz replacing E. Nelson for TRW.

2. GENERAL BUSINESS

- The AIE environment development is still in a hold status. The compiler development continues with completion expected in November 1984 and validation in January 1985.

- The ALS development continues with compiler validation expected in September 1984. The interim environment has been distributed by the Army to selected applicants for rehosting.

- The ALS/N procurement for the standard Navy environment has been delayed.

- The CSC contract for development of a Configuration Management tool has expired. A report presenting the results of a Configuration Management Workshop held under this contract will be published in the next Public Report.

- Texas Instruments has completed design of the AIM tool but implementation is delayed pending additional compiler validations.

3. WORKING GROUP REPORTS

- KITIA Working Group 1 expects to devote 70% of their time to review of the Requirements and Criteria (RAC) document and 30% of their time to review of the Common APSE Interface Set (CAIS) document.

- KITIA Working Group 3 will retain their review role and address policy issues that arise.

- The RACWG has a new version of the RAC document available for review. They have received inputs from Working Group 1 and solicit additional inputs.

- The CAISWG is awaiting additional comments from the services as part of the review process of Version 1.1. The NTIS order number for this document is A134825.

- The DEFWG has been working on a list of terms for a standard Glossary.

- The STANDWG is investigating procedures to meet MIL-STD document formats. They are preparing a cross-reference for specifications that may impact the CAIS document.

- The COMPWG met in Washington in December. They have prepared a matrix approach to requirements verification for the RAC and CAIS documents which will be presented later in the meeting.

- The GACWG is reviewing an outline for a User's Guide to Transportability.

4. MILNET/TAC ACCESS

- The MILNET and ARPANET are splitting. The ARPANET TAC will remain the same but the MILNET TAC will require a new login procedure. Each user must have a registered login number. Contact the host administrator to register. For ECLB users it is MARK@ECLB.

5. GENERAL BUSINESS

- There will be a KIT presentation at the next Tri-Service review.
- The next Public Report is expected to be completed at the end of April 1984. Public Report #3 is now at the printer for reproduction and should be available shortly.
- Rudy Krutar pointed out that Tartan Labs is not tasked to maintain DIANA. NRL is looking at a task on intermediate language standardization.
- Bruno Witte (NOSC) is working on building an Ada math library and will release an RFP shortly. As part of this process he will conduct a survey to identify desired items for this math library. The proposals for WWMCCS Ada tools are due. The main goal of this procurement was to use a number of companies to construct a quick and dirty toolset.

6. CAIS SCHEDULE

- The revised schedule for generation of the CAIS document is:
  - Version 1.2 April 84 (response to comments submitted)
  - Version 1.3 November 84 (version for review by services)
  - Version 1.4 January 85 (MIL-STD 1)
  - Tri-Service Configuration Control Board January 85
  - Version 2 start January 85 (for calendar year 86 review cycle)
  - MIL-STD 2 in January 87
- The CAISWG will determine the schedule for replies to comments.

7. FUTURE MEETING SCHEDULE

- Future KIT/KITIA meeting are planned for
  - Apr 9-12 Seattle
  - Jul 16-19 Toronto
  - Oct 1-4 New Hampshire

  1985
  - Jan San Diego
  - Apr Washington D.C.
  - Jul San Francisco
  - Sep Connecticut/Texas/United Kingdom (?)

  1986
  - Jan San Diego

8. BREAK FOR NUMBERED WORKING GROUP MEETINGS

9. RECONVENE INTO SEPARATE KIT AND KITIA MEETINGS

10. KIT DoD ONLY MEETING

11. BREAK FOR LUNCH

12. RECONVENE FOR NAMED WORKING GROUP MEETINGS

13. REQUIREMENTS AND CRITERIA DOCUMENT PRESENTATIONS

- George Robertson presented a format for analysis of the R&C document utilizing a matrix format. Requirements and criteria on one axis can be matched to CAIS features on the other axis. This is intended to provide traceability from the R&C to the CAIS.

- A general discussion regarding the database support requirements for the CAIS occurred. Major issues included the use of unique names and version groups. Version groups are called for in the STONEMAN document. No real database resolutions were attained.

- Frank Belz gave a presentation on different levels of CAIS layering and the associated functionality of each layer's application. A general discussion of the pros and cons for a layered CAIS followed.

14. BREAK FOR DAY

18 JANUARY 1984

15. REQUIREMENTS AND CRITERIA DOCUMENT DISCUSSIONS

- An open discussion of the R&C document sections was conducted. It was repeated that the R&C document is intended to apply to CAIS Version 2 which will be submitted for designation as a Military Standard and is not binding on the current Version 1.1 or expected Version 1.2 document.

- Emphasis was made that the R&C requires item-by-item review to insure that solutions (design) are segregated from requirements.

- Each section of the document needs its own Introduction.

16. BREAK FOR LUNCH

17. OBJECT MANAGEMENT SUPPORT

- A proposed revision for the Object Management Support section of the R&C document was circulated to the KIT/KITIA for review. Following a general discussion, the proposed changes were deferred pending further analysis. KIT/KITIA members wished additional time to review the recommendations.

18. BREAK FOR NUMBERED WORKING GROUPS

19. BREAK FOR DAY

19 JANUARY 1984

20. EVALUATION AND VALIDATION PRESENTATION

- Ginny Castor presented the plans and status for the newly formed APSE Evaluation and Validation Team. The team had its first meeting in December and is planning to sponsor a yearly E&V Workshop to solicit inputs from industry and academia. Attendance will be by invitation based on a submitted point paper. The first workshop is scheduled for April 1984. The KIT/KITIA will be able to interchange data since some of the KIT members are also on the team.

21. TECHNICAL PRESENTATIONS

- Tim Lindquist presented a methodology for identifying and correcting some of the consistency and completeness problems in the initial CAIS. His approach is to describe the CAIS by syntax functionality, protocol, and limits through the use of an abstract machine.

- Roy Freedman is preparing a methodology for the specification of CAIS semantics via operational semantics. This is expected to provide an operational definition of interoperability and transportability by example.

- Henry Lefkowitz presented results of work performed by Alpha Omega Group on a Data Dictionary System. Some of their conclusions include data collection at a single point of entry, the system must support programmers, it must provide management information, must have automatic data capture, and the database must be available for query during programming.

22. BREAK FOR LUNCH

23. RECONVENE INTO NAMED WORKING GROUPS

24. RAC SCHEDULE

- The schedule for revision and review of the R&C document for vote by the KIT and KITIA was distributed. A revised document will be available by 19 March 1984.

25. KITIA POLICY STATEMENT

- Herm Fischer presented to the KIT the draft KITIA Policy Statement regarding APSE distribution as Government Furnished Equipment on contracts. This position will be presented to the next Tri-Service Review for consideration. The KITIA presented its position to the KIT for information purposes.

26. MEETING ADJOURNED

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
16-19 January 1984

KIT Attendees:

BELZ, Frank	TRW
CASTOR, Jinny	AFWAL/AAAF
CHADWICK, Kevin	Canadian National Defense HQ
FERGUSON, Jay	DoD
FOIDL, Jack	TRW
FOREMAN, John	Texas Instruments
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
HOLLISTER, John	CECOM
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LEE, Elizabeth	NSWC/DL
LINDLEY, Larry	NAC
LOPER, Warren	NOSC
MOLONEY, Jim	Intermetrics
MILLER, Jo	NWC
MYERS, Gil	NOSC
MYERS, Phillip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
ROBERTSON, George	FCDSSA-SD
SCHAAR, Brian	AJPO

TAFT, Tucker	Intermetrics
TAYLOR, Guy	FCDSSA-DN
THALL, Rich	SofTech
WALTRIP, Chuck	John Hopkins Univ.
WILDER, Bill	PMS-408

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas Astronautics
BRYAN, Doug	Lockheed Missiles & Space
CORNHILL, Dennis	Honeywell/SRC
COX, Fred	Georgia Institute of Technology
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Sytsems
FISHER, Gerry	CSC
FREEDMAN, Roy	Hazeltine Corp.
GLASEMAN, Steve	Aerospace Corp.
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles & Space
LAMB, J. Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
MORSE, H. R.	Frey Federal Systems
PLOEDEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
RUBY, Jim	Hughes Aircraft Co.
RUDMIK, Andres	GTE
SAIB, Sabina	General Research Corp.
SIBLEY, Edgar	Alpha Omega Group, Inc.
WESTERMANN, Rob	TNO-IBBC The Netherlands
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

APPENDIX B - MEETING HANDOUTS

1. Position paper, The Need for a CAIS Operational Semantics, R.S. Freedman

2. Department of Defense Trusted Computer System Evaluation Criteria, DoD, 15 August 1983.

3. "Security Requirements for Ada Programming Support Environments" DRAFT, David C. Pasterchik, dtd. 16 January 1984.

4. "Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for the Common APSE Interface Set" WORKING PAPER, dtd. 13 January 1984.

5. "Ada Environments as GFE is Harmful", D.E. Wrege, dtd. 13 January 1984

6. KITIA Policy Recommendations, Draft, dtd. 17 January 1984.

SECTION 3

KIT/KITIA DOCUMENTATION

Ada Programming Support Environment (APSE)

Interoperability and Transportability (I&T) Management Plan

January 1984

for

Ada JOINT PROGRAM OFFICE
The Pentagon
Washington, D.C. 20301

prepared by

NAVAL OCEAN SYSTEMS CENTER
271 Catalina Boulevard
San Diego, California 92152

TABLE OF CONTENTS

1.0 INTRODUCTION

The Ada Programming Support Environment (APSE) Interoperability and Transportability (I&T) Plan is presented in this document. The I&T activities necessary to achieve sharing of tools and data bases between APSEs are described. Schedules and milestones for these activities are presented as well as a Work Breakdown Structure (WBS) for accomplishing them.

These I&T activities are conducted by the Kernel APSE Interface Team (KIT).

The major responsibilities are:

a. APSE I&T Management
b. APSE I&T Analysis
c. APSE I&T Standards Development
d. APSE I&T Tools Development
e. APSE I&T Coordination with Implementation Efforts

1.1 BACKGROUND

In 1975 the High Order Language Working Group (HOLWG) was formed under the auspices of the U.S. Department of Defense (DoD) with the goal of establishing a single high order language for new DoD Embedded Computer Systems (ECS). The technical requirements for the common language were finalized in the Steelman report [1] of June 1978. International competition was used to select the new common language design. In 1979 the DoD selected the design developed by Jean Ichbiah and his colleagues at CII-Honeywell Bull. The language was named Ada in honor of Augusta Ada Byron (1816-1851), the daughter of Lord Byron and the first computer programmer.

It was realized early in the development process that acceptance of a common language and the benefits derived from a common language could be increased substantially by the development of an integrated system of software development and maintenance tools. The requirements for such an Ada programming environment were stated in the STONEMAN document [2]. The STONEMAN paints a broad picture of the needs and identifies the relationships of the parts of an integrated APSE. STONEMAN identifies the APSE as support for "the development and maintenance of Ada application software throughout its life cycle". The APSE is to provide a well-coordinated set of tools with uniform interfaces to support a programming project throughout its life cycle. The Initial Operational Capabilities (IOCs) are called Minimal Ada Programming Support Environments (MAPSEs).

[1] Requirements For High Order Computer Programming Languages: STEELMAN, DoD, June 1978

[2] Requirements for Ada Programming Support Environments, STONEMAN, DoD,

The Army and Air Force have begun separate developments of APSEs. The Army APSE has been designated the ALS (Ada Language System) and that of the Air Force, the AIE (Ada Integrated Environment). The Navy APSE will make maximum use of those Army and Air Force products that meet Navy requirements and will require the development of only those additional components required for Navy applications.

The Ada Joint Program Office (AJPO) was formed in December 1980. The AJPO coordinates all Ada efforts within DoD to ensure their compatibility with the requirements of other Services and DoD agencies, to avoid duplicative efforts, and to maximize sharing of resources. The AJPO is the principal DoD agent for development, support and distribution of Ada tools and Ada common libraries.

1.2 DEFINITIONS

INTEROPERABILITY: Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.

TRANSPORTABILITY: Transportability of an APSE tool is the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonyms.

1.3 OBJECTIVES

The objectives of the APSE I&T effort are:

a. To develop requirements for APSE I&T.

STONEMAN paints a broad picture of the needs and relationships of the parts of an integrated APSE. Although STONEMAN is being used as the primary requirements document for APSE development efforts, it does not provide sufficient detail to assure I&T between APSEs. APSEs built to accomodate I&T requirements will insure cost savings in the development of tools. The cost of reprogramming tools for different APSEs will be significantly reduced.

b. To develop guidelines, conventions and standards to be used to achieve I&T of APSEs.

Guidelines, conventions, and standards describe the means by which the requirements can be satisfied. It would be premature to develop steadfast standards during the early part of this APSE I&T effort. There is little precedent for I&T between programming support environments of this anticipated magnitude and thus little guidance for the development of these guidelines, conventions, and standards. The guidelines, conventions and standards that are developed during this APSE I&T effort will evolve over a five year period from 1982 through 1987. These guidelines, conventions, and standards will be presented in public forums to insure that they are sound and realistic.

c. To develop APSE I&T tools to be integrated into both the AIE and ALS.

This APSE I&T effort provides for the development of tools to be integrated into both the AIE and the ALS. These tool development efforts will help identify interfaces and surface interface problems associated with I&T between different APSEs. They should also show how closely the guidelines, conventions and standards developed by this APSE I&T effort reflect the reality of the AIE and ALS efforts. But the tools developed by this APSE I&T effort will not be limited to this test function. They will also be well documented tools which will become useful additions to any APSE.

d. To monitor the AIE and ALS development efforts with respect to APSE I&T.

This APSE I&T effort provides for the monitoring of the AIE and ALS development efforts. The monitoring will result in recommendations for resolution of differences between the AIE or the ALS and the evolving APSE I&T conventions and standards. Interface areas which would inhibit I&T between the AIE and ALS will also be identified.

AIE and ALS documents will be reviewed and analyzed, and recommendations will be made. When questions arise that need resolution and/or clarification with regard to the ALS and AIE development efforts the KIT (see Section 2.3) will rely on the assistance of Army and Air Force members who are involved in these efforts.

e. To provide initiative and give a focal point with respect to APSE I&T.

A focal point is needed for APSE developers and users with regard to information about I&T. APSE I&T questions arise frequently within professional societies and user groups. A forum is needed in which APSE I&T questions can be addressed and discussed and in which APSE I&T information can be disseminated throughout the Ada community.

The KIT and KITIA (see Sections 2.3 and 2.4) will provide focal points for the Ada community. Public reports on the results of this APSE I&T effort will be published every six months. This is in keeping with the AJPO philosophy of public exposure of all aspects of the Ada program. The KIT and KITIA will also participate in other programs connected with APSE I&T, including international development efforts, whenever possible.

f. To develop and implement procedures to determine compliance of APSE developments with APSE I&T requirements, guidelines, conventions and standards.

Procedures must be established by which the recommendations that are developed by this APSE I&T effort will be reviewed and implemented by the AJPO. The procedures that are to be followed should apply not only to the AIE and ALS development efforts, but also to other APSE development efforts. Work on the determination of compliance procedures will be pursued in cooperation with the AJPO's Evaluation and Validation program.

1.4 DOCUMENT ORGANIZATION

Section 1 of this document discusses the purpose and scope of the I&T Plan, the objectives of the I&T effort, and the basic concepts, definitions, and objectives.

Section 2 discusses the sponsorship, the participating organizations, the organizational inter-relationships and responsibilities, and the potential forums for public involvement.

The specific tasks to be accomplished in persuit of I&T are covered in Section 3. These functions are presented in a work breakdown structure for the project and a schedule of milestones and deliverables.

Special needs in achieving I&T are discussed in Section 4.

Appendix A contains a glossary of terms and acronyms applicable to the I&T effort and Appendix B contains a bibliography of AIE documents. Appendix C contains a bibliography of ALS documents and Appendix D contains other APSE related documentation. Appendix E describes the elements of the I&T Work Breakdown Structure.

2.0 ORGANIZATION

Figure 1 shows the participants in the APSE I&T effort. The following sections provide a brief description of these organizations and their relationships.

2.1 Ada JOINT PROGRAM OFFICE

The KIT is an agent of the Ada Joint Program Office (AJPO). The KIT supports the AJPO by performing the activities outlined in this plan and by providing recommendations and information to the AJPO. The AJPO makes final decisions in the areas of requirements, policy, procedures and funding.

2.2 ARMY, AIR FORCE AND NAVY

Currently the Army and Air Force have begun separate developments of APSEs. In the development of its APSE, the Navy plans to make maximum use of Army/Air Force products that meet Navy requirements. The KIT will review of all these APSE developments and identify critical aspects of the designs where conventions or standard interfaces and specifications are needed to insure compatibility. It will be the role of the KIT to interact with these services and their respective APSE contractors for information-exchange and consultation. The contractor for the Army's ALS is SofTech Inc.; the Air Force contractor for the AIE is Intermetrics Inc.. The Navy contractor has not been selected yet. Representatives of both the Air Force and Army APSE development efforts are members of the KIT, and many members of the Navy's Design Review Group (DRG) serve on the KIT as well.

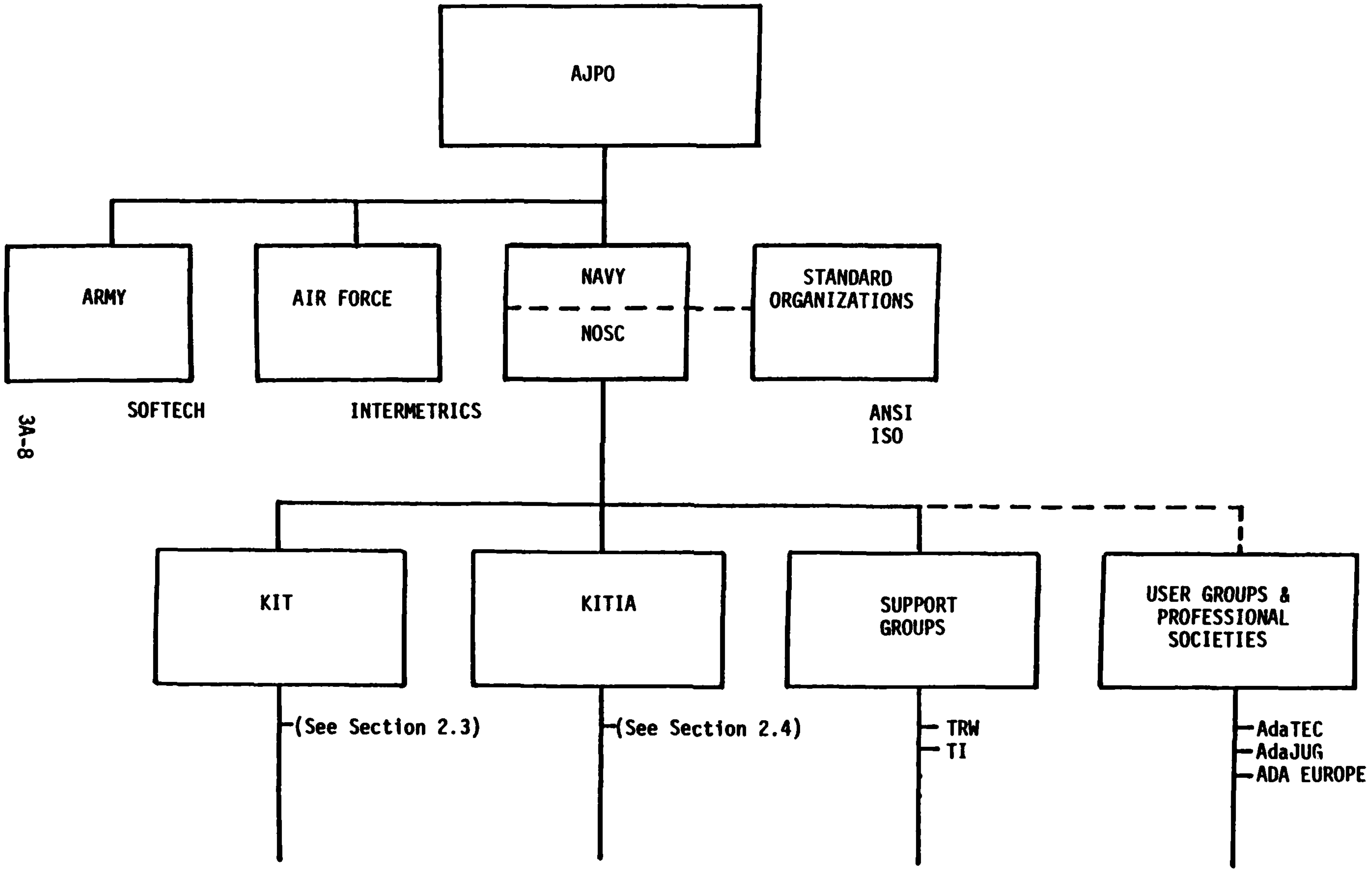


Figure 1 APSE IT Participants

2.3 KAPSE INTERFACE TEAM (KIT)

The objectives of the KIT are the objectives of the APSE I&T effort (see Section 1.3). The Navy is responsibile for chairing the KIT. The membership is composed of the following DoD representatives:

- o Navy Deputy to the Ada Joint Program Office
- o Naval Ocean Systems Center (NOSC)
- o Naval Sea Systems Command (NAVSEA/PMS-408)
- o Naval Electronics Systems Command (NAVELEX)
- o Naval Underwater Systems Center (NUSC)
- o Naval Surface Weapons Center (NSWC)
- o Naval Avionics Center (NAC)
- o Naval Air Development Center (NADC)
- o Naval Research Laboratory (NRL)
- o Naval Weapons Center (NWC)
- o Fleet Combat Direction System Support Activity (FCDSSA) - Dam Neck
- o Fleet Combat Direction System Support Activity (FCDSSA) - San Diego
- o U.S. Air Force - Rome Air Development Center (RADC)
- o U.S. Air Force - Air Force Wright Aeronautical Laboratories (AFWAL)
- o U.S. Army - Communications and Electronics Command
- o U.S. Air Force - Information Processing Standards for Computers (USAF-IPSC)
- o Johns Hopkins University Applied Physics Laboratory (JHUAPL)
- o National Security Agency
- o Canadian National Defense Headquarters

NOSC is the Navy laboratory which provides the KIT chairman. All other members participate on a volunteer basis, aided as necessary by the AJPO with funding for such things as travel expenses. New members will be added to the KIT at the discretion of the AJPO.

Because of the potentially large membership of the KIT, a management steering committee called the KIT Executive Committee (KITEC) has been established. It consists of the AJPO sponsor (i.e., the AJPO Navy deputy), the KIT chairman, the primary support contractor (see Section 2.5), and selected other KIT members as determined by the sponsor and chairman. The KITEC is responsible for the planning and management of the APSE I&T effort, including maintenance of this plan and direction of activities in accordance with its tasks and schedules.

In addition, the KIT is divided into various working groups for the purpose of small group concentration on specific technical areas affecting I&T. The number, objectives, and memebership of such working groups may change as KIT needs change.

2.4 KAPSE INTERFACE TEAM FROM INDUSTRY AND ACADEMIA

The KITIA was formed to complement the KIT and to generally contribute a non-DoD perspective to the I&T effort. The KITIA supplements the activities of the KIT. It assures broad inputs from software experts and eventual users of APSE's. The KITIA interacts with the KIT as reviewers, as proposers of APSE I&T requirements, guidelines, conventions and standards, and as consultants concerning implementation implications. The team was selected from applicants representing industry and academia. The following are the members of the KITIA:

Alpha-Omega Group
Aerospace Corporation
Bell Laboratories
Boeing Aerospace
Computer Sciences Corporation
Control Data Corporation
Ford Aerospace
Frey Federal Systems
General Electric
General Research
General Telephone & Electronics Laboratories
Georgia Institute of Technology
Grumman Aerospace
Hazeltine
Honeywell
Hughes Aircraft
IABG (W. Germany)
IBM
Litton
Lockheed
McDonnell Douglas
Norden
PRC
Raytheon
SDC
Teledyne
TNO (The Netherlands)
UK Ada Consortium
Virginia Polytechnic Institute

In addition, the following has been asked to be a special associate member of the team:

Oy Softplan Ab (Finland)

Membership on the team belongs to a company or university and not to an individual representing his/her organization. All participation is voluntary, and the members selected have agreed to provide 1/3 of a man-year plus other support such as travel expenses. The membership of the KITIA will not be expanded unless an organization withdraws or very special circumstances apply. The AJPO sponsor and KIT chairman are ex officio members of the KITIA.

The KITIA elects a chairman and a vice-chairman from amongst its participants every year. It, too, is organized into working groups who in turn select their own chairmen. The KITIA chairman and vice-chairman together with the working group chairmen form the KITIA management committee.

The KITIA is responsible to the AJPO through the KIT chairman. Although the KIT has ultimate responsibility for the development of all products required to meet the I&T objectives, the KITIA participates directly in the generation and review of such products. In addition, the KITIA generates its own contributing papers, products, initiatives, and recommendations to supplement and guide the basic KIT efforts. This requires close coordination, which is facilitated by ARPANET communication mechanisms, parallel working group structures, and joint team meetings.

2.5 SUPPORT CONTRACTORS

Currently there are three contractors that participate on the KIT. TRW is the primary support contractor, providing general support and technical initiatives. Texas Instruments is developing an APSE tool in support of the I&T objectives (see Section 1.3c). SoftWrights provides overall review and consultation for the AJPO.

Any of these contractors may also serve as a vice-chairman of a KIT working group.

2.6 USER GROUPS AND PROFESSIONAL SOCIETIES

It is anticipated that AdaTEC, the Ada-JOVIAL Users Group (Ada-JUG), and Ada Europe will provide valuable contributions to the APSE I&T effort. The KIT and KITIA have no formal relationship with these groups; however, the KITEC will use some or all of these groups as regular forums for the presentation of reports and technical results and will solicit feedback from their members.

2.7 STANDARDS ORGANIZATIONS

The American National Standards Institute (ANSI) and the International Standards Organization (ISO) are standards organizations which are already involved in establishing the Ada programming language as a broadly recognized, enforceable standard. It is possible that the results of this I&T effort will be submitted for such approval by these organizations as well, to effect the commonality of APSE's deemed necessary to achieve DoD's life-cycle objectives. The KIT initially will become familiar with the organizations' standardization procedures so that future standardization actions can be planned and accomplished with minimum difficulty. This will include the study of existing standards which may interact with or guide the development of APSE I&T standards.

2.8 LIAISON WITH IMPLEMENTATION EFFORTS

A number of implementation efforts have been undertaken by organizations outside of the DoD. Three of these (the U.K. Ada Consortium, the West German IABG and U.C. Irvine) have participated on the KITIA. Others include the European Economic Community, ROLM Corporation, Western Digital, and Telesoft, just to name a few. The KIT will keep such organizations informed of its activities and will consider all feedback received from them.

3.0 APSE I&T PLAN

This section shows the Work Breakdown Structure (WBS) for the I&T effort as well as the schedules and milestones for the WBS elements. Figures 2 thru 7 provide an overview for the WBS elements. Figure 8 provides a summary of the schedule.

3.1 WORK BREAKDOWN STRUCTURE

A discussion of the major elements in the WBS is presented below. Detailed task descriptions are contained in Appendix E.

1000 APSE Interoperability & Transportability Management

This WBS element covers the general management tasks required to accomplish the APSE I&T objectives. It includes general project and team management, project planning, general meeting and team support and configuration management.

2000 APSE Interoperability & Transportability Analysis

This WBS element covers the technical analysis tasks required to accomplish the APSE I&T objectives. It includes resource reviews, requirements development, and performance of special studies.

3000 APSE Interoperability & Transportability Standards

This WBS element describes the standardization tasks required to accomplish the APSE I&T objectives. It includes guidelines and conventions development, specification development, compliance and validation formulation, common APSE interface set analysis, and definition of the standardization process.

4000 APSE Interoperability & Transportability Tools

This WBS element describes the development of APSE tools that support the APSE I&T objectives. It includes planning and acquisition of tools, tool development, test and analysis, and maintenance and modification of developed tools.

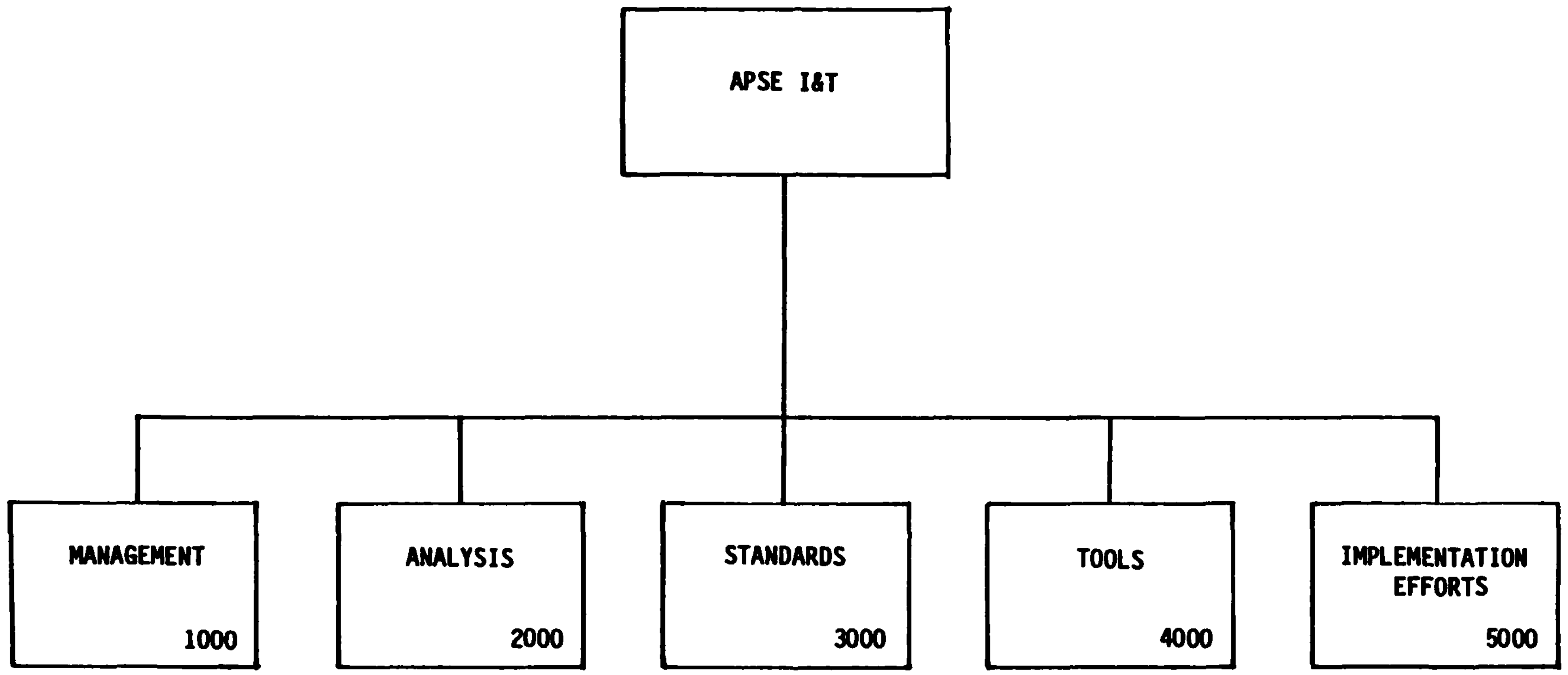


Figure 2. WBS Overview

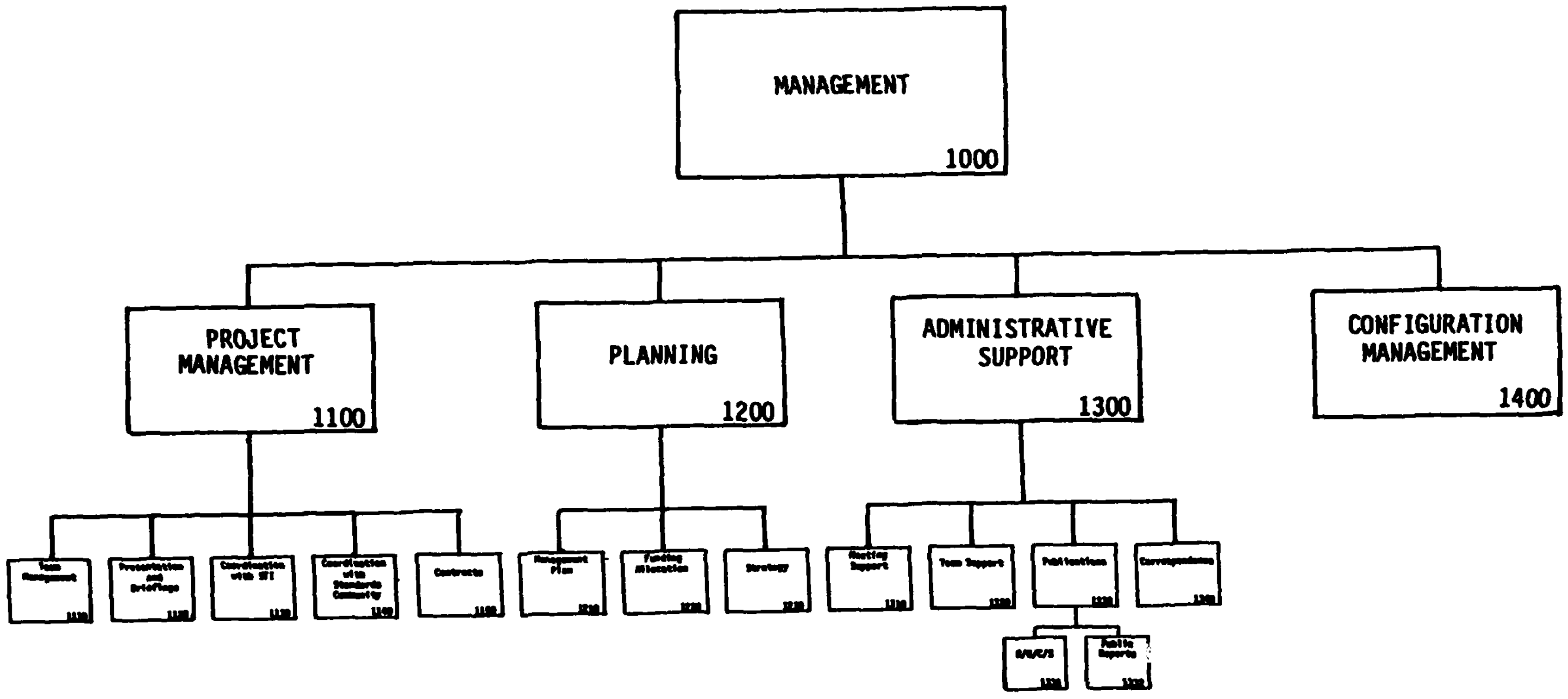


Figure 3. WBS Management Overview

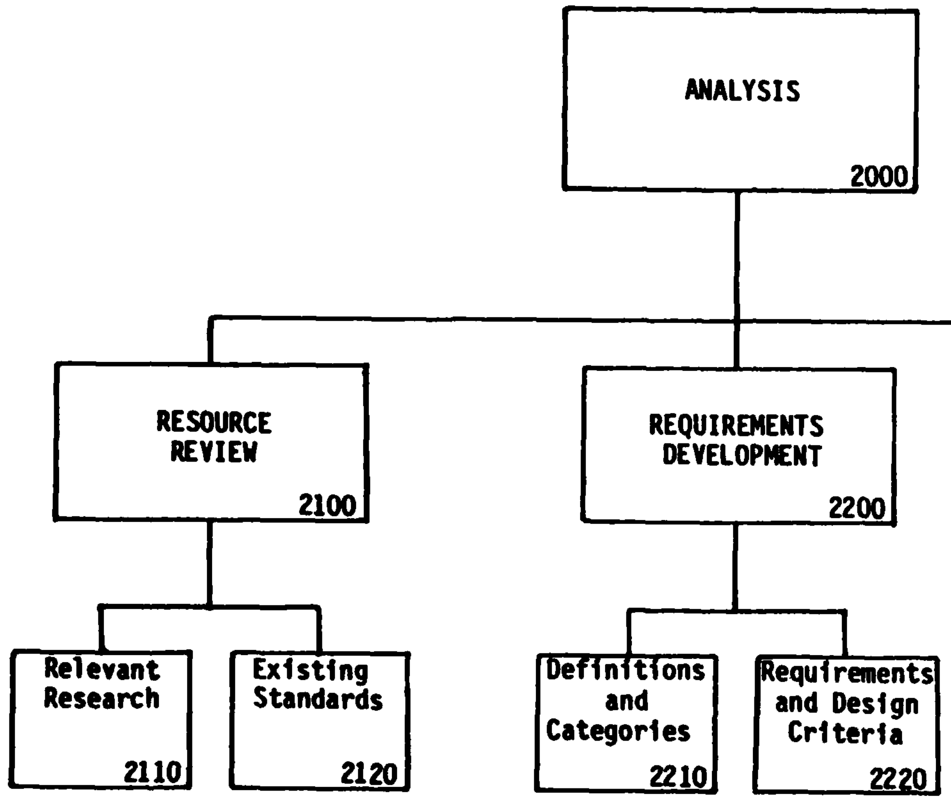


Figure 4. WBS Analysis Overview

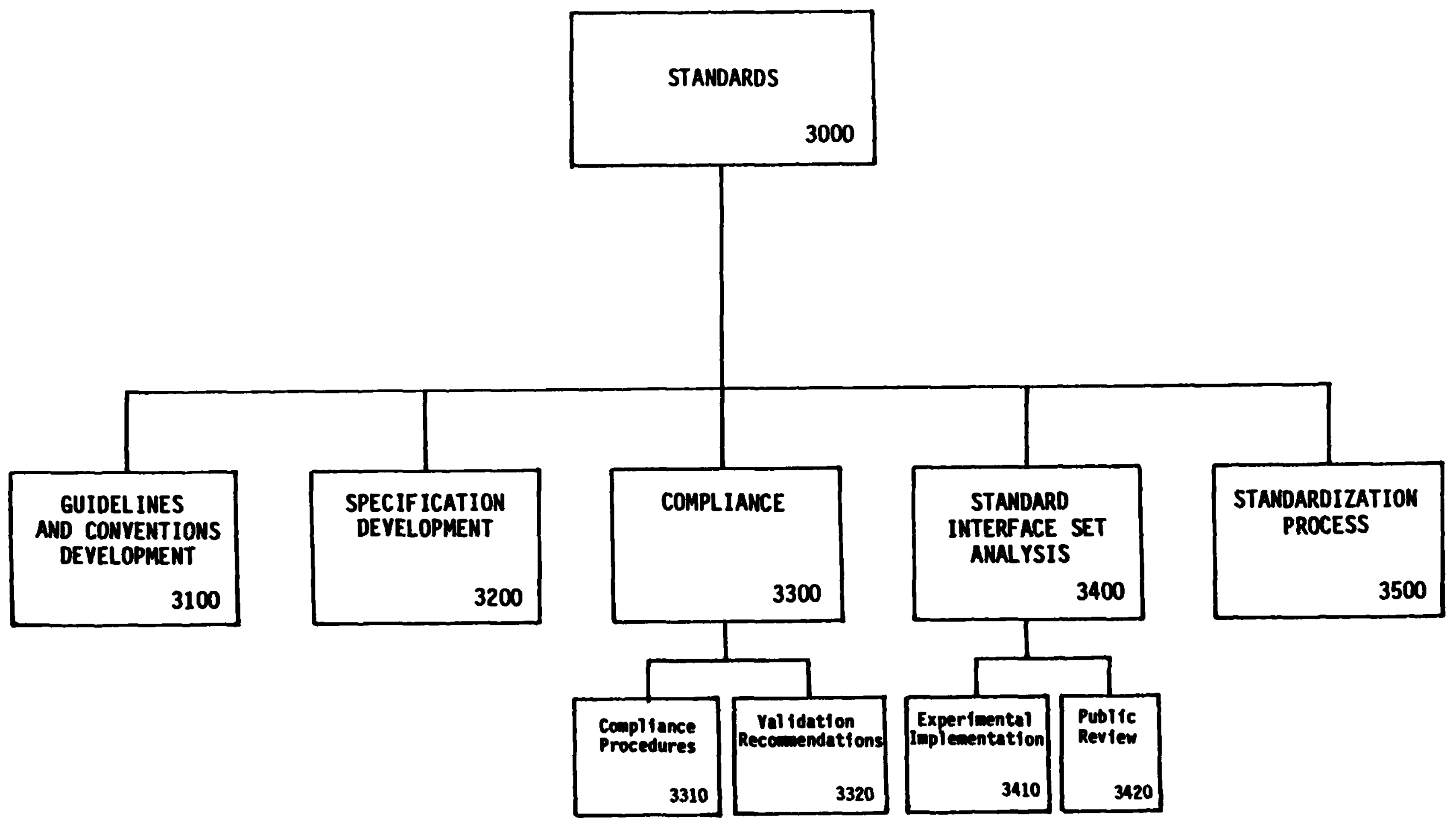

STANDARDS 3000
GUIDELINES AND CONVENTIONS DEVELOPMENT 3100
SPECIFICATION DEVELOPMENT 3200
COMPLIANCE 3300
Compliance Procedures 3310
Validation Recommendations 3320
STANDARD INTERFACE SET ANALYSIS 3400
Experimental Implementation 3410
Public Review 3420
STANDARDIZATION PROCESS 3500

Figure 5. WBS Standards Overview

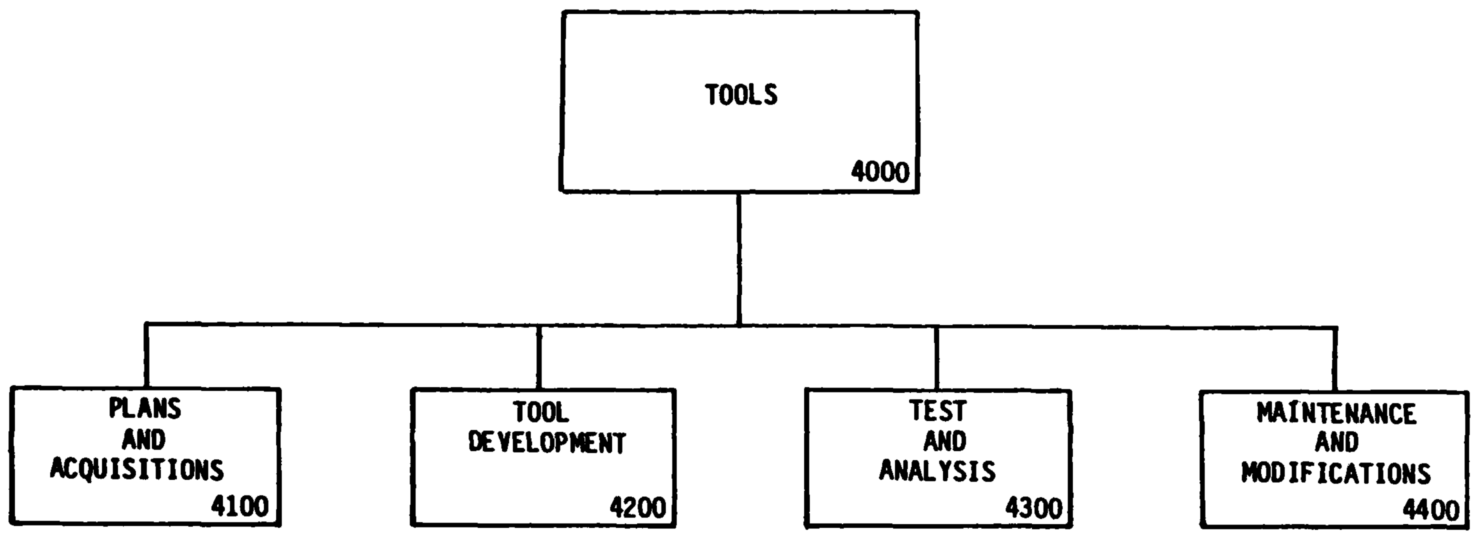


Figure 6. WBS Tools Overview

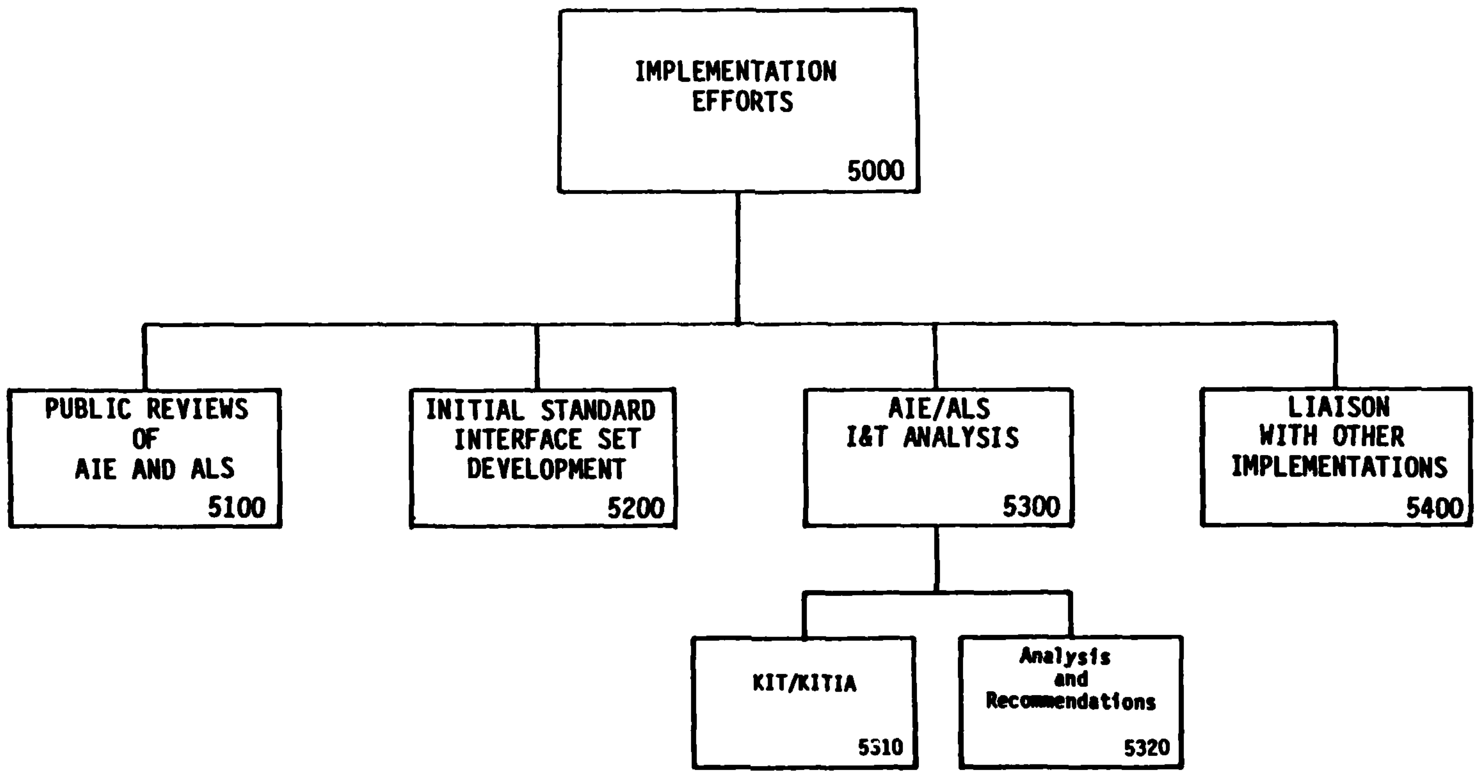
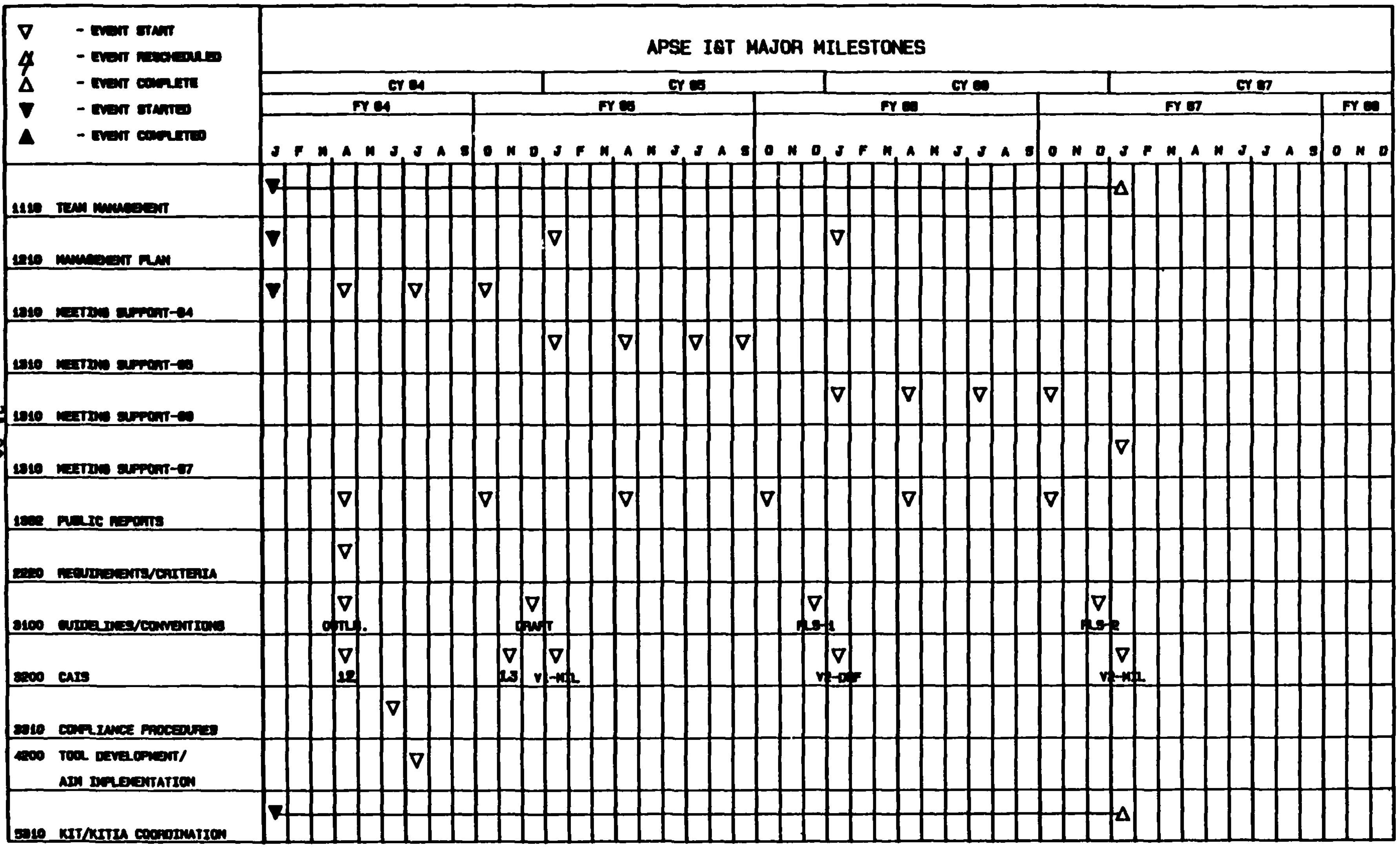

IMPLEMENTATION EFFORTS
5000
PUBLIC REVIEWS OF AIE AND ALS
5100
INITIAL STANDARD INTERFACE SET DEVELOPMENT
5200
AIE/ALS I&T ANALYSIS
5300
LIAISON WITH OTHER IMPLEMENTATIONS
5400
KIT/KITIA
5310
Analysis and Recommendations
5320

Figure 7. WBS Implementation Effort Overview


APSE I&T MAJOR MILESTONES
▽ - EVENT START
- EVENT RESCHEDULED
△ - EVENT COMPLETE
▼ - EVENT STARTED
▲ - EVENT COMPLETED
CY 84
CY 85
CY 86
CY 87
FY 84
FY 85
FY 86
FY 87
FY 88
J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A S O N D
1110 TEAM MANAGEMENT
1210 MANAGEMENT PLAN
1310 MEETING SUPPORT-84
1310 MEETING SUPPORT-85
1310 MEETING SUPPORT-86
1310 MEETING SUPPORT-87
1302 PUBLIC REPORTS
2220 REQUIREMENTS/CRITERIA
3100 GUIDELINES/CONVENTIONS
3200 CAIS
3310 COMPLIANCE PROCEDURES
4200 TOOL DEVELOPMENT/ AIM IMPLEMENTATION
5310 KIT/KITIA COORDINATION
OUTLN.
DRAFT
RLS-1
RLS-2
1.2
1.3
V1-MIL
V2-DEF
V2-MIL

5000 APSE Coordination with Implementation Efforts

This WBS element describes the tasks affecting various APSE development efforts required to support the APSE I&T objectives. It includes public reviews of the AIE and ALS, development of an initial Common APSE Interface Set, I&T analysis of AIE and ALS, and liaison with other implementations.

4.0 PROVISIONS FOR SPECIAL NEEDS

This APSE I&T Plan emphasizes the development of requirements, conventions and standards. It is unusual in that it is written for a programming language support environment that is in the development state. At this point in development it is essential for the KITEC to provide an I&T forum and act as a focal point for the Ada community, APSE developers and the DoD. This will provide broad input to the KIT from which a complete, realistic set of I&T requirements, guidelines, conventions and standards will be developed that respond to ongoing APSE development and long term APSE needs.

Normally to achieve APSE I&T the APSE itself would be written in Ada. However, STONEMAN recognizes that "in cases where there is a large current investment in software projects, written originally in other languages", provisions and guidelines must be developed that account for cost effective transitions to Ada environments. In the development of APSE I&T requirements, conventions, and standards the KITEC should provide cost benefit analysis with respect to their recommendations and decisions concerning implementation.

During the initial phase of carrying out this APSE I&T plan the KITEC will be studying and contributing to the I&T aspects of APSE developments by the Army and Air Force (ALS and AIE). When the Navy begins its development of an APSE the KITEC will also concern itself with the I&T aspects of its design. The KITEC will develop requirements, conventions, and standards that can be used for validation testing. In addition APSE development by the private sector and international development will be addressed. Criteria for validation testing for all APSE development efforts should be established. In the future a central agent can perform I&T validation testing on each APSE. The model for a strong central validation capability is the Ada Compiler Validation Facility.

APPENDIX A

GLOSSARY

GLOSSARY OF TERMS

Ada-JUG	Ada-JOVIAL Users Group
AIE	Ada Integrated Environment
AJPO	Ada Joint Program Office
ALS	Ada Language System
ANSI	American National Standards Institute
APSE	Ada Programming Support Environment
DIANA	Descriptive Intermediate Attributed Notation for Ada
DoD	Department of Defense
ECS	Embedded Computer System
FCDSSA	Fleet Combat Direction System Support Activity
GCS	Guidelines, Conventions and Standards
HOLWG	High Order Language Working Group
IOC	Initial Operational Capabilities
ISO	International Standards Organization
I&T	Interoperability and Transportability
JCL	Job Control Language
JHUAPL	John Hopkins University Applied Physics Laboratory
KAPSE	Kernel Ada Programming Support Environment
KIT	KAPSE Interface Team
KITIA	KAPSE Interface Team from Industry and Academia
KITEC	KAPSE Interface Team Executive Committee
MAPSE	Minimal Ada Programming Support Environment
MOA	Memorandum of Agreement
NAC	Naval Avionics Center
NADC	Naval Air Development Center
NAVELEX	Naval Electronic Systems Command
NAVSEA	Naval Sea Systems Command
NOSC	Naval Ocean Systems Center
NRL	Naval Research Laboratory
NSWC	Naval Surface Weapons Center
NUSC	Naval Underwater Systems Center
NWC	Naval Weapons Center
RFP	Request For Proposal
WBS	Work Breadkdown Structure

APPENDIX B

AIE DOCUMENTS

APPLICABLE DOCUMENTS

The following documents are important sources of information relevant to the KIT effort. While the list does not represent a comprehensive bibliography on the subject of standardization, interoperability, and transportability it does constitute information sources essential to the project.

AIE Documents

- System Specification for Ada Integrated Environment, Intermetrics, November, 1982
- Computer Program Test Plan, Intermetrics, December, 1982
- Ada Integrated Environment Computer Program Development Plan, Intermetrics, November, 1982
- Computer Program Development Specification for Ada Integrated Environment:

  Virtual Memory Methodology, Intermetrics, October, 1982
  Ada Compiler Phases, Intermetrics, November, 1982
  KAPSE/Database, Intermetrics, November, 1982
  MAPSE Command Processor, Intermetrics, December, 1982
  MAPSE Debugging Facilities, Intermetrics, January, 1983
  Program Integration Facilities, Intermetrics, August, 1983

- Computer Program Interface Specification for Ada Integrated Environment:

  Dianna, Intermetrics, December, 1982
  Bill, Intermetrics, December, 1982

- Computer Program Product Specification for Ada Integrated Environment:

  Virtual Memory Methodology, Intermetrics, November, 1982
  Ada Compiler Front End, Intermetrics, November, 1982
  Ada Compiler Middle End for the IBM 4341, Intermetrics, January, 1983
  Ada Compiler Back End for the IBM 4341, Intermetrics January, 1983
  KAPSE Run-Time Support, Intermetrics, February, 1983

KAPSE Simple and Composite Objects, Intermetrics, February, 1983

Program Integration Facility: Program Builder, Intermetrics, November, 1983

- Computer Program Test Procedures for Ada Integrated Environment:

  Virtual Memory Methodology, Intermetrics, November, 1982
  Ada Compiler Front End, Intermetrics, December, 1982
  Ada Compiler Middle Part, Intermetrics, January, 1983
  Ada Compiler Back End, Intermetrics, January, 1983
  KAPSE Run-Time Support, Intermetrics, May, 1983
  Program Builder, Intermetrics, November, 1983

APPENDIX C

ALS DOCUMENTS

APPLICABLE DOCUMENTS

The following documents are important sources of information relevant to the KIT effort. While the list does not represent a comprehensive bibliography on the subject of standardization, interoperability, and transportability it does constitute information sources essential to the project.

ALS Documents

- ALS Retargeting Manual, SofTech, November, 1983
- ALS Operators Manual, SofTech, November, 1983
- ALS Specification, SofTech, November, 1983
- ALS KAPSE - B5 Specification, SofTech, November, 1983
- ALS VAX-11/780 Linker - B5 Specification, SofTech, November, 1983
- ALS VAX-11/780 Code Generator - B5 Specification, SofTech, November, 1983
- ALS VAX-11/780 Assembler - B5 Specification, SofTech, November, 1983
- ALS VAX/VMS Linker - B5 Specification, SofTech, November 1983
- ALS VAX/VMS Symbolic Debugger - B5 Specification, SofTech, November 1983
- ALS VAX/VMS Frequency Analyzer - B5 Specification, SofTech, November 1983
- ALS VAX/VMS Statistical Analyzer - B5 Specification, SofTech, November 1983
- ALS Command Language Processor - B5 Specification, SofTech, November 1983
- ALS Database Manager - B5 Specification, SofTech, November 1983
- ALS Configuration Control Tools - B5 Specification, SofTech, November 1983
- ALS KAPSE - B5 Specification, SofTech, November 1983
- ALS Machine Independent Section - B5 Specification, SofTech, November 1983
- ALS File Administrator - B5 Specification, SofTech, November 1983
- ALS Display Tools - B5 Specification, SofTech, November 1983
- ALS VAX/VMS Code Generator (Vol. I & II) - C5 Specification, SofTech, November 1983
- ALS VAX/VMS Runtime Support Library (Vol. I & II) - C5 Specification, SofTech, November 1983
- ALS VAX/VMS Assembler - C5 Specification, SofTech, November 1983
- ALS VAX/VMS Linker - C5 Specification, SofTech, November 1983

- ALS VAX/VMS Symbolic Debugger (Vol. I & II) - C5 Specification, SofTech, November 1983
- ALS VAX/VMS Frequency Analyzer - C5 Specification, SofTech, November 1983
- ALS VAX/VMS Statistical Analyzer - C5 Specification, SofTech, November 1983
- ALS Command Language Processor - C5 Specification, SofTech, November 1983
- ALS Database Manager (Vol. I & II) - C5 Specification, SofTech, November 1983
- ALS Configuration Control Tools - C5 Specification, SofTech, November 1983
- ALS KAPSE - C5 Specification, SofTech, November 1983
- ALS Machine Independent Section (Vol. I, II & III) - C5 Specification, SofTech, November 1983
- ALS File Administrator - C5 Specification, SofTech, November 1983
- ALS Display Tools (Vol. I & II) - C5 Specification, SofTech, November 1983
- ALS Interim Ada-to-Pascal Translation Tool Language Reference Manual, SofTech, January 1984
- ALS Coding Conventions and Naming Rules Manual, SofTech, January 1984
- ALS Regeneration Kit Manual, SofTech, January 1984

APPENDIX D

OTHER DOCUMENTS

Other Documents

- Requirements For High Order Computer Programming Languages: STEELMAN, DoD, June 1978
- Requirements for Ada Programming Support Environments, STONEMAN, DoD, February 1980
- Interface Analysis of the Ada Integrated Environment and the Ada Language System, J.M. Foidl, TRW, October 1982.
- Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report, Volume I, Naval Ocean Systems Center, Technical Document 509, 1 April 1982.

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1110	WBS ELEMENT TITLE: Team Management
PART OF WBS ELEMENT: 1100 - Project Management	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: NOSC Code 8322 with KITIA Chairman support	
TASK DESCRIPTION: Assemble original teams. Coordinate the solicitation and selection of new members. Organize team structure into working groups. Coordinate KIT and KITIA activities separately and together. Organize and coordinate all team meetings. Assign team and working group tasks and see to their completion. Plan and chair meetings. Coordinate the raising and resolution of issues.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1120	WBS ELEMENT TITLE: Presentations and Briefings

PART OF WBS ELEMENT: 1100 Project Management

DELIVERABLES/MILESTONES:

Project Review	May 1983
Senior Management Brief	Summer 1983, Spring 1984
AdaTec Conference	October 1983 October 1984 October 1985

RESPONSIBILITY: NOSC Code 8322 with AJPO support

TASK DESCRIPTION: Prepare slides and narration on team objectives, status, progress and plans. Present materials at project reviews, senior management briefings, AdaTEC conferences, symposia, etc.

NOTES:

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1130	WBS ELEMENT TITLE: Coordination with Software Technology Initiative (STI)
PART OF WBS ELEMENT: 1100 Project Management	
DE'.IVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: NOSC Code 8322 with AJPO	
TASK DESCRIPTION: Attend STI workshops. Cooperate with STI personnel to assure proper incorporation of KIT/KITIA work into STI plans.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1140	WBS ELEMENT TITLE: Coordination with Standards Community

PART OF WBS ELEMENT: 1100 Project Management

DELIVERABLES/MILESTONES: Continuous

RESPONSIBILITY: NOSC Code 8322 with AJPO support

TASK DESCRIPTION: Keep standards community apprised of team activities and progress, primarily through cooperation with Bill LaPlant, IPSC. Submit descriptions and reports as requested. Locate and track relevant standards activities.

NOTES:

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 1150	WBS ELEMENT TITLE: Contracts
PART OF WBS ELEMENT: 1100 Project Management	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: NOSC Code 8322	
TASK DESCRIPTION: Initiate contracts and/or tasking necessary to achieve project objectives. At minimum, this includes contracts for tools and general support. Monitor progress including reviews and examination of deliverables. Coordinate the incorporation of results of contracts into general KIT/KITIA work.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1210	WBS ELEMENT TITLE: Management Plan

PART OF WBS ELEMENT: 1200 Planning

DELIVERABLES/MILESTONES:

APSE I&T Management Plan	April 1983
	January 1984
	January 1985

RESPONSIBILITY: NOSC Code 8322

TASK DESCRIPTION: Plan activities as necessary to complete the APSE I&T project. Document all plans in the APSE I&T Management Plan. Update this plan once a year, or more often if radical changes occur.

NOTES: An earlier version of this plan was published as CDRL Item A001 of Delivery Order #7N45 on TRW Contract N00123-80-D-0242.

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1220	WBS ELEMENT TITLE: Funding Allocation
PART OF WBS ELEMENT: 1200 Planning	
DELIVERABLES/MILESTONES: Budget updates quarterly	
RESPONSIBILITY: NOSC Code 8322	
TASK DESCRIPTION: Establish budget for project activities. Secure funds as required. Manage the distribution and expenditure of funds by NOSC, contractors and other agencies. Update budget as necessary.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1230	WBS ELEMENT TITLE: Strategy
PART OF WBS ELEMENT: 1200 Planning	
DELIVERABLES/MILESTONES: APSE I&T Implementation Strategy May 1983	
RESPONSIBILITY: NOSC Code 8322	
TASK DESCRIPTION: Establish, plan and document the strategy to be followed by KIT/KITIA in pursuit of APSE I&T objectives. Reflect this strategy in all plans, budgets and task assignments.	
NOTES:	

WBS ELEMENT DESCRIPTION
ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983
REVISION:
REVISION DATE:

WBS ELEMENT NR: 1310	WBS ELEMENT TITLE: Meeting Support
PART OF WBS ELEMENT: 1300 Administrative Support	
DELIVERABLES/MILESTONES: All support is required quarterly in conjunction with regular KIT/KITIA meetings. Other support is also required for special meetings and some working group activities.	
RESPONSIBILITY: Support Contractor with NOSC Code 8322.	
TASK DESCRIPTION: Provide technical support required in planning, preparing for, conducting and reporting on APSE I&T meetings. Support includes, but is not limited to, the provision of agendas, discussion copies of papers, meeting arrangements, minutes and attendee lists.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1320	WBS ELEMENT TITLE: Team Support
PART OF WBS ELEMENT: 1300 Administrative Support	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: Support Contractor with NOSC Code 8322	
TASK DESCRIPTION: Provide technical support required for maintenance, storage, updating and distribution of documents and data of the APSE I&T project. Support includes, but is not limited to, maintenance of address lists, document control, working paper preparation and ARPANET directory administration, such as for KIT-INFORMATION and various comment directories.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1331	WBS ELEMENT TITLE: Requirements, Guidelines, Conventions and Standards

PART OF WBS ELEMENT: 1330 Publications/1300 Administrative Support

DELIVERABLES/MILESTONES:

Requirements	December 1983
Guidelines/Conventions	June 1985
Standard	December 1984 December 1985

RESPONSIBILITY: NOSC Code 8322 with Support Contractor

TASK DESCRIPTION: Generate final versions of all named documents. Submit them to all appropriate publication processes. Provide for their distribution to the KIT/KITIA and to the public through NTIS.

NOTES:

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 1332

WBS ELEMENT TITLE: Public Reports

PART OF WBS ELEMENT: 1330 Publications/1330 Administrative Support

DELIVERABLES/MILESTONES:

Public Report Vol. III	April 1983
Public Report Vol. IV	October 1983
Public Report Vol. V	April 1984
Public Report Vol. VI	October 1984
Public Report Vol. VII	April 1985
Public Report Vol.VIII	October 1985

RESPONSIBILITY: NOSC Code 8322 with Support Contractor

TASK DESCRIPTION: Generate publishable versions of all public reports. This includes determination and acquisition of contents, reformatting as necessary, organization, submission to publication process, distribution, notification of report availability and maintenance of the notification addressee list. Public distribution will be through NTIS.

NOTES:

WBS ELEMENT DESCRIPTION — ORIGINAL DATE: 30 April 1983

ORIGINATOR: NOSC — REVISION:

REVISION DATE:

WBS ELEMENT NR: 1340	WBS ELEMENT TITLE: Correspondence
PART OF WBS ELEMENT: 1300 Administrative Support	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Conduct communications as necessary, particularly using the ARPANET. NOSC requirements in this element include the provision of terminals, ports and other required facilities in support of NOSC's other tasks.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 1400	WBS ELEMENT TITLE: Configuration Management
PART OF WBS ELEMENT: 1000 APSE I&T Management	
DELIVERABLES/MILESTONES: Configuration Management Report December 1983	
RESPONSIBILITY: NOSC Code 8322 with Support Contractor	
TASK DESCRIPTION: Plan for configuration management of tools developed under this project. Perform configuration management during the project.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 2110	WBS ELEMENT TITLE: Relevant Research
PART OF WBS ELEMENT: 2100 Resource Reviews	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Review literature and documentation applicable to I&T requirements, guidelines, conventions and standards.	
NOTES:	

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 2120	WBS ELEMENT TITLE: Existing Standards
PART OF WBS ELEMENT: 2100 Resource Reviews	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Locate and examine relevant standards. Use and/or incorporate relevant standards as found to be appropriate and applicable.	
NOTES: As an example of this, the Operating System Command and Response Language (OSCRL) User Requirements, Functional Requirements and Design Criteria have been used as models for the APSE I&T Requirements and Criteria. The OSCRL documents were developed by X3H1.	

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 2210	WBS ELEMENT TITLE: Definitions and Categories
PART OF WBS ELEMENT: 2200 Requirements Development	
DELIVERABLES/MILESTONES: KAPSE Interface Worksheets December 1983	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Develop definitions of all relevant terms, particularly "interoperability" and "transportability". Develop categories of interfaces and KAPSE Interface Worksheets describing each of them.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 2220	WBS ELEMENT TITLE: Requirements and Design Criteria

PART OF WBS ELEMENT: 2200 Requirements Development

DELIVERABLES/MILESTONES:

Requirements and Criteria	December 1983

RESPONSIBILITY: All participants

TASK DESCRIPTION: Develop functional requirements and interface design criteria for a set of interfaces which will achieve APSE I&T. Document and analyze these requirements and criteria. Analysis will be conducted through public review as well as team review and will determine completeness, consistency and feasibility.

NOTES:

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 2300	WBS ELEMENT TITLE: Special Studies
PART OF WBS ELEMENT: 2000 APSE I&T Analysis	
DELIVERABLES/MILESTONES: Workshops and reports as appropriate Command Language workshop and report Configuration Management workshop, June 1983 Configuration Management report, June 1983	
RESPONSIBILITY: Various participants	
TASK DESCRIPTION: Conduct technical analyses and studies as required. These special studies may include such topics as command languages, configuration management, STONEMAN revision and risks and cost benefits associated with various levels of I&T.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3100	WBS ELEMENT TITLE: Guidelines and Conventions Development
PART OF WBS ELEMENT: 3000 APSE I&T Standards	
DELIVERABLES/MILESTONES: APSE I&T Guidelines and Conventions Review Draft April 1984	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Develop guidelines and conventions for achieving I&T. These supplement and further explain the standard, covering those ideas and approaches that have not been included in the standard as yet but which are believed to contribute to the achievement of I&T.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3200	WBS ELEMENT TITLE: Specification Development

PART OF WBS ELEMENT: 3000 APSE I&T Standards

DELIVERABLES/MILESTONES:

Standard Interface Set Specification Review Draft	December 1984

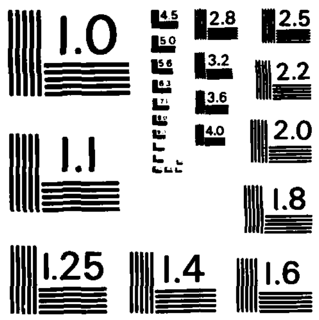
RESPONSIBILITY: NOSC Code 8322 and all participants

TASK DESCRIPTION: Develop the set of interface specifications which will be recommended to the AJPO for standardization. Review and analyze these with respect to conformance with the requirements and criteria and to consistency, completeness and feasibility.

NOTES:

AD-A147 648 KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT VOLUME 4(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 30 APR 84 2/5

UNCLASSIFIED NOSC/TD-552-VOL-4 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3310	WBS ELEMENT TITLE: Compliance Procedures
PART OF WBS ELEMENT: 3300 Compliance	
DELIVERABLES/MILESTONES: Compliance Procedures June 1984	
RESPONSIBILITY: NOSC Code 8322 with Support Contractor	
TASK DESCRIPTION: Develop procedures for determining compliance of an APSE implementation with APSE I&T requirements, guidelines, conventions and standards. Apply these procedures experimentally to the I&T tools and the AIE and ALS. The results of this task will influence the form the standard specification will take.	
NOTES: This compliance work will be conducted in close cooperation with the AJPO Evaluation and Validation team and will form the basis of the KIT/KITIA's recommendations to this team.	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3320	WBS ELEMENT TITLE: Validation Recommendations
PART OF WBS ELEMENT: 3300 Compliance	
DELIVERABLES/MILESTONES: Validation Recommendations December 1985	
RESPONSIBILITY: NOSC Code 8322 with Support Contractor	
TASK DESCRIPTION: Review the results of the development and application of the Compliance Procedures (WBS 3310). Formulate recommendations for the AJPO and its Evaluation and Validation team.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3410	WBS ELEMENT TITLE: Experimental Implementation

PART OF WBS ELEMENT: 3400 Standard Interface Set Analysis

DELIVERABLES/MILESTONES:

Implementation Report	June 1985

RESPONSIBILITY: NOSC Code 8322 with Support Contractor

TASK DESCRIPTION: Experimentally implement and exercise portions of the proposed standard interface set in order to investigate feasibility, completeness, etc. Report results as feedback to be incorporated in final standard interface set specification.

NOTES:

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 3420	WBS ELEMENT TITLE: Public Review
PART OF WBS ELEMENT: 3400 Standard Interface Set Analysis	
DELIVERABLES/MILESTONES: Complete Review June 1985	
RESPONSIBILITY: NOSC Code 8322	
TASK DESCRIPTION: Present the proposed standard for widespread public review, including an open review meeting. Incorporate all feedback in final documents.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 3500	WBS ELEMENT TITLE: Standardization Process
PART OF WBS ELEMENT: 3000 APSE I&T Standards	
DELIVERABLES/MILESTONES: Initiate effort June 1985	
RESPONSIBILITY: NOSC Code 8322 with AJPO	
TASK DESCRIPTION: Determine steps required to achieve standardization of the proposed interface set. Pursue standardization.	
NOTES: This activity alone among all these tasks may be expected to continue beyond the lifetime of the KIT/KITIA.	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 4100	WBS ELEMENT TITLE: Plans and Acquisition

PART OF WBS ELEMENT: 4000 APSE I&T Tools

DELIVERABLES/MILESTONES:

Plans	July 1982
Acquisition	October 1983

RESPONSIBILITY: NOSC Code 8322

TASK DESCRIPTION: Identify the objectives, criteria and requirements to be used for the selection of three or more APSE tools. These tools will be used to further analyze interface requirements. Initiate acquisition of three or more such tools.

NOTES:

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 4200	WBS ELEMENT TITLE: Tool Development

PART OF WBS ELEMENT: 4000 APSE I&T Tools

DELIVERABLES/MILESTONES:

CMS Design	June 1983
AIM Implementation	June 1984
Others	NLT December 1985

RESPONSIBILITY: Selected Contractors

TASK DESCRIPTION: Design, develop and test tools in a local environment. Install and integrate tools in both the AIE and ALS. Provide insights into interface issues as they arise during development and integration.

NOTES:

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 4300	WBS ELEMENT TITLE: Test and Analysis

PART OF WBS ELEMENT: 4000 APSE I&T Tools

DELIVERABLES/MILESTONES: Test Reports June 1985

RESPONSIBILITY: NOSC Code 8322 with Support Contractor

TASK DESCRIPTION: Develop test applications and analyses for determining performance of APSE I&T tools in the AIE and ALS. Apply these to tools as they are completed.

NOTES:

WBS ELEMENT DESCRIPTION

ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983

REVISION:

REVISION DATE:

WBS ELEMENT NR: 4400	WBS ELEMENT TITLE: Maintenance and Modifications
PART OF WBS ELEMENT: 4000 APSE I&T Tools	
DELIVERABLES/MILESTONES: As required	
RESPONSIBILITY: NOSC Code 8322 and Contractors	
TASK DESCRIPTION: Provide maintenance of APSE I&T tools as required after their integration into AIE and ALS. Modify tools in accordance with needs to correct inadequacies or to respond to changing requirements or environments.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 5100	WBS ELEMENT TITLE: Public Reviews of AIE and ALS

PART OF WBS ELEMENT: 5000 APSE I&T Coordination with Implementation Efforts

DELIVERABLES/MILESTONES:

Public Review Reports	July 1982 (ALS)
	July 1983 (AIE)
	January 1984 (ALS)
	July 1984 (AIE)
	January 1985 (ALS)
	July 1985 (AIE)

RESPONSIBILITY: NOSC Code 831

TASK DESCRIPTION: Coordinate the establishment and notification of review teams. Determine documents or systems to be reviewed and arrange for distribution of copies to members of review teams. Receive all team review reports and correlate into report to AJPO and AIE/ALS sponsor.

NOTES:

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 5200	WBS ELEMENT TITLE: Initial Standard Interface Set Development
PART OF WBS ELEMENT: 5000 APSE I&T Coordination with Implementation Efforts	
DELIVERABLES/MILESTONES: Initial SIS Draft Report June 1983	
RESPONSIBILITY: Selected participants with NOSC Code 8322	
TASK DESCRIPTION: Review AIE and ALS to determine a set of interfaces which is implementable in both of these systems. Develop a specification report documenting these interfaces. This task is to be accomplished with participation of AIE and ALS personnel.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 5310	WBS ELEMENT TITLE: KIT/KITIA Coordination
PART OF WBS ELEMENT: 5300 AIE/ALS I&T Analysis	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: NOSC Code 8322	
TASK DESCRIPTION: Provide channels of communication between KIT/KITIA members and government and contractor personnel involved in the AIE and ALS developments. Arrange for meetings and distribution of relevant documents. Provide feedback to AIE and ALS developers.	
NOTES:	

WBS ELEMENT DESCRIPTION	ORIGINAL DATE: 30 April 1983
ORIGINATOR: NOSC	REVISION:
	REVISION DATE:

WBS ELEMENT NR: 5320	WBS ELEMENT TITLE: Analysis and Recommendations
PART OF WBS ELEMENT: 5300 AIE/ALS I&T Analysis	
DELIVERABLES/MILESTONES: Analysis Report January 1985	
RESPONSIBILITY: Various participants	
TASK DESCRIPTION: Analyze AIE and ALS interfaces with respect to I&T. Provide recommendations for evaluation of each system to meet the interface set as it is put forward for standardization.	
NOTES:	

WBS ELEMENT DESCRIPTION
ORIGINATOR: NOSC

ORIGINAL DATE: 30 April 1983
REVISION:
REVISION DATE:

WBS ELEMENT NR: 5400	WBS ELEMENT TITLE: Liaison with Other Implementations
PART OF WBS ELEMENT: 5000 APSE I&T Coordination with Implementation Efforts	
DELIVERABLES/MILESTONES: Continuous	
RESPONSIBILITY: All participants	
TASK DESCRIPTION: Maintain awareness of and contact with groups who are doing non-DoD APSE implementations. Solicit their inputs and provide information on KIT/KITIA activities. Examples of such groups are the UK, IABG in W. Germany, the EEC, ROLM and UC Irvine.	
NOTES:	

Draft Specification of the Common APSE Interface Set (CAIS)

Version 1.1
30 September, 1983

Prepared by

KIT/KITIA
CAIS Working Group

for the
Ada® Joint Program Office

(® Ada is a Registered Trademark of the Department of Defense, Ada Joint Program Office)

FORWORD

This document is a draft for public review. It will be revised in accordance with comments received during this public review cycle.

This document has been prepared in response to the Memorandum of Agreement signed by the Undersecretary of Defense and the Assistant Secretaries of the Air Force, Army, and Navy. The memorandum established agreement for defining a set of common interfaces for the Department of Defense (DoD) Ada Programming Support Environments (APSEs) to promote Ada tool transportability and interoperability. The initial phase of this effort is directed toward the interfaces of the Ada Integrated Environment (AIE) and the Ada Language System (ALS). This version derives a set of specific interfaces from these two APSEs, but the CAIS is intended to be implementable as part of a wide variety of intended APSEs. It is anticipated that the CAIS will evolve, changing to meet new needs. Ultimately it is the intention of the DoD to submit CAIS for standardization. Through the acceptance of such a standard it is anticipated that the source level compatability of Ada software tools will be enhanced for both the DoD and non-DoD users.

The authors of this document include technical representatives of the two DoD APSE contractors, representatives from the DoD's Kernel Ada Programming Support Environment (KAPSE) Interface Team (KIT), and volunteer representatives from the KAPSE Interface Team, Industry and Academia (KITIA).

The initial effort for definition of the CAIS was begun in September 1982 by the following members of the KAPSE Interface Team (KIT): J. Foidl (TRW), J. Kramer (Ada Joint Program Office), T. Oberndorf (Naval Ocean Systems Center), T. Taft (Intermetrics), R. Thall and W. Wilder (both of SofTech). In February 1983 the design team was expanded by Lcdr. B. Schaar (Ada Joint Program Office) to utilize the professional capabilities and experience of the KIT and KAPSE Interface Team from Industry and Academia (KITIA). These new members include: H. Fischer (Litton Data Systems), T. Harrison (Texas Instruments), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Frey Federal Systems), E. Ploedereder (I.A.B.G., West Germany), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace). The Ada Joint Program Office is particularly grateful to those KITIA members and their companies for providing the time and resources that significantly contributed to this document. Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center) and the general memberships of the KIT and KITIA.

This document was typeset by McMahon Engineering Services, San Diego, using a Compugraphics MCS20-8400 typesetting unit, with the Advanced Communication Interface used to transfer data already keystroked from a word processor to the Compugraphics typesetter and then inputting typesetting codes to format the document.

CONTENTS

1. INTRODUCTION

This document provides specifications for a set of Ada packages which together form a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs). This interface set is designed to promote the source-level portability of Ada programs, particularly Ada software development tools. The initial phase of this effort is directed toward the interfaces of the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Version 1.1 of the CAIS, presented herein, is intended to provide the basis for evolution of the CAIS as APSEs are implemented, as tools are transported, and as tool interoperability issues are encountered.

Tools written in Ada, using only the packages described herein, should be transportable to other CAIS implimentations. However, where tools function as a set, the CAIS facilitates transportability of the set of tools as a whole, but individual tools may not be individually transportable.

1.1 SCOPE OF THE CAIS

This version of the CAIS establishes interface requirements for the transportability of Ada toolsets software to be utilized in Department of Defense (DoD) Ada Programming Support Environments (APSEs) known as the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Strict adherence to this interface set will ensure that the Ada toolsets will possess the highest degree of portability across APSEs.

The scope of the CAIS includes interfaces to those services traditionally provided by an operating system that affect tool transportability. Ideally, all APSE tools would be implementable using only the Ada language and the CAIS. This version of the CAIS is intended to provide most interfaces required by common tools. This version of the CAIS includes six interface areas:

a. **Node Model.** This area presents a node model for the CAIS in which contents, relationships and attributes of nodes are defined. Also included are the foundations for access control and synchronization.

b. **Structural Nodes.** This area covers the creation of structural nodes.

c. **File Nodes.** This area covers file input/output.

d. **Process Nodes.** This area covers creation of processes for program invocation, control of processes, process attribute management, and inter-process communication.

e. **Device Nodes.** This area covers basic device input/output support, along with special device control facilities.

f. **Utilities.** This area covers text and list manipulation.

1.2 EXCLUDED AND DEFERRED TOPICS

During the design of the CAIS many aspects of environments have been considered. It has been determined that several aspects should be explicitly excluded from this version of the CAIS:

Interfaces for non-software development environments (target systems) are not a part of this version.

The acronyms KAPSE and MAPSE are not used in this document because there is disagreement on their meanings.

Multi-lingual environments are not addressed by the CAIS.

A number of interface issues remain unresolved in this version of the CAIS, even though they have been considered. These issues are important for a complete interface specification, but their resolution has been deferred until a later version. Deferred interface issues (in alphabetical order) include:

Access control — Access rights and privileges to system resources.

Asynchronous interfaces — Most interfaces in this document are task synchronous interfaces (i.e., the specified operation is completed before the calling task is allowed to proceed.)

Communications transformation — filtering of data before receipt by processes, mappings (lower case to upper case, break, key to escape sequence), terminator character for input.

Configuration management — configuration control including keeping versions, referencing the latest revision, identifying the state of an object, etc.

Device control — Controls for printers, tape drives, disk drives, graphics, windowing, etc.

Distributed environments — Explicit support for environments in which parts of Ada programs or data bases are distributed across multiple processors.

Interoperability — Inter-tool interfaces for tool sets; calling sequences and data formats used to invoke/interact with common APSE tools, including the compilation/program library system, the text editing systems, the command processor, and the mail system.

Predefined attributes/names — A full set of attributes and names that exist in all APSEs which implement the CAIS.

Predefined exceptions — A full set of exceptions that exist in all APSEs which implement the CAIS; identification of all situations where exceptions are raised by the CAIS.

Resource access and management — Resource control and allocation, such as for processor time, processor memory, and shared data pools.

Security — Mechanisms for handling discretionary and non-discretionary information based on classification of the data and system requirements.

Typed database — Typing of the objects in the database organization.

1.3 CONFORMANCE

Conformance of an implementation to the CAIS is established on a package-by-package basis. Each package must be available as a library unit, with the name specified in this document. From the package user's point of view, the package must have indistinguishable syntax and semantics from those stated herein. The following differences in CAIS package implementation from the specifications in this document are considered indistinguishable from a user's point of view:

a) The package may have additional WITH or USE clauses.

b) Parameter modes listed here as OUT may be IN OUT or those listed as IN OUT may be OUT.

c) Types specified as limited private may be simply limited types.

d) Packages may be instantiations of generic sub-packages of some other (private) library unit package.

Examples of differences which are NOT legal:

a) Additional or missing declarations, as these affect name visibility.

b) Parameter mode IN OUT, as this prevents passing of expressions.

c Limited private types being changed to sub-types or derived types, when this changes the semantics of "deriving" from the type.

d) Packages which are not available as specified library units, because this changes the means of reference to package components.

1.4 DOCUMENT ORGANIZATION

Each of the interface areas described in Section 1.1 is the subject of a subsequent section of this document. A discussion introduces the underlying model for that area. Ada package specifications describe the facilities provided. These are followed by a narrative of the intended semantics of the package. New terms introduced in the narrative sections of the CAIS have been highlighted with **boldface** type. **Boldface** type within the package specifications and package semantics sections indicate reserved words in accordance with the Ada Language Reference Manual.

2. REFERENCES

[LRM]: Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A; United States Department of Defense; January 1983.

[STONEMAN]: Requirements for Ada Programming Support Environments, "Stoneman"; Department of Defense; February 1980.

KERNEL Ada Programming Support Environment (KAPSE) Interface Team: Public Report; Volume I, Naval Ocean Systems Center TD509; April 1982.

APSE Interoperability and Transportability Implementation Strategy; Ada Joint Program Office; June 1983.

[ANSI 79]: American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)"; July 1979.

[ANSI 77]: American National Standards Institute, "American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)"; June 1977.

ALS KAPSE — B5 Specification, SofTech; February 1982.

Computer Program Development Specification for Ada Integrated Environment: KAPSE/DATABASE TYPE B5, Intermetrics Inc.; 12 Nov 1982.

3. CAIS NODE MODEL

The CAIS implementation acts as a manager for a set of entities that may be files, processes, and devices. These entities have properties and may be interrelated in many ways,

The CAIS model uses the notion of a **node** as a carrier of information about an entity. It uses the notion of a **relationship** for representing an interrelation between entities and the notion of an **attribute** for representing a property of an entity or of an interrelation.

This version of the CAIS identifies four different kinds of nodes: structural nodes, file nodes, process nodes, and device nodes.

The structure provided by the CAIS node model is a directed graph of nodes, each of which may have content, relationships and attributes; relationships may also have attributes. The **content** varies with the kind of node. If a node is a structural node, there is no content and the node is used strictly as a holder of relationships and attributes. If a node is a file node, the content is an Ada external file. If a node is a process node, the content is the representation of the execution of an Ada program. If a node is a device node, its content is a representation of a logical or physical device.

3.1 RELATIONSHIPS AND RELATIONS

The relationships of CAIS nodes form the edges of a directed graph; they are used to build conventional hierarchical directory and process structures (see Section 4.1 CAIS_STRUCTURAL_NODES, Section 6.2 CAIS_PROCESS_CONTROL and Appendix B) as well as arbitrary directed-graph structures. Relationships are unidirectional and are said to emanate from a **source node** and to terminate at a **target node**.

Because any node may have many relationships representing many different classes of connections, the concept of a **relation** is introduced to categorize the relationships. These relations identify the nature of relationships, and **relationships** are instances of relations. There are several predefined relations provided by the CAIS. These are: PARENT, USER, JOB, CURRENT_JOB, CURRENT_USER, CURRENT_NODE, and DOT and are explained in the following sections.

Each relationship is designated by a relation name and a relationship key. The **relation name** identifies the relation and the **relationship key** distinguishes between multiple nodes each bearing the same relation with a given node. If a relationship is a unique instance of its relation (i.e., only one node bears the relation with a given node), the key may be omitted (i.e., its value is the null string). In this document, a relation name is often referred to simply as a relation and a relationship key is often referred to simply as a key. Nodes in the environment are accessible by navigating along the (named) relationships. Operations are provided to move from one node (along one of its relationships) to a connected node.

3.1.1 Kinds of Relationships

There are two kinds of relationships: primary and secondary. Primary relationships form a strict tree; secondary relationships may form an arbitrary directed graph. There is no requirement that all primary relationships have the same relation name.

When a node is created, a primary relationship must be initially established from some other node, called its **parent** node. This initial relationship is marked as the **primary** relationship for this new node. As a side effect of the creation, the new node will be connected back to this parent via the PARENT relation (which, because it is unique, has a null relationship key). To delete a node, the primary relationship is broken. RENAME (see Section 3.5) may be used to make the primary relationship emanate from a different parent. These operations maintain a state in which each non-root node has exactly one parent and a unique primary pathname (see Section 3.1.2).

Secondary relationships are arbitrary connections which may be established between two existing nodes. They are created with the LINK procedure (see Section 3.5) and broken with the UNLINK procedure. If a node is deleted (i.e., its primary relationship is broken), outstanding secondary relationships for which it is the target may remain, but attempts to access the node via these relationships will raise an exception.

3.1.2 Predefined Relations

The CAIS node model incorporates the notion of a user. Each user has one top-level node (often called the user directory). This top-level node is the root of the user's work-area tree, and from it he can access other structural, file, process and device nodes. Every node may be accessed by following a sequence of relationships; this sequence is called the **path** to the node. A path starting at a top-level node is called an **absolute** path. Every node can be traced back to its top-level node by recursively following PARENT relationships; the path obtained by inverting this chain is the unique primary path to the node.

A path can also start at a known (not necessarily top-level) node and follow a sequence of relationships to a desired node. This is a **relative** path and the known starting node is called the **base**.

Any user's top-level node can be accessed from a proces node using the relation USER and a relationship key which is interpreted as the user's name. User names may in fact be names of projects, services, people, or other organizational entities; each has a top level node associated with it. It is anticipated that certain special user names will be defined (as an eventual part of the CAIS) to provide uniform access to common tools, structures, etc.. Each implementation must identify such user names to be of special significance in the environment.

When a user enters the APSE, a root process node is created which often represents a command interpreter or other user-communication process; a process tree develops from this root node as other processes are invoked for the user. A particular user may have entered the APSE several times concurrently. Each corresponding process tree is referred to as a job. The JOB relation is provided for locating each of these root processes from the user's top-level node. Thus a JOB relation emanates from each user's top-level node to the root process node of each of the user's jobs. The JOB relation must always be used with a relationship-key which identifies the name of the particular job which is to be accessed.

Any process node in a job has associated with it at least three predefined relations. The CURRENT_JOB relationship always points to the root node for a process node's job. The CURRENT_USER relationship always points to the user's top-level node. The CURRENT_NODE relationship always points to a node which represents the process node's current focus or context for its activities; the target node is often a structural node. The process node can thus use the CURRENT_NODE for a base node when specifying relative paths. All three of these relations (CURRENT_JOB, CURRENT_USER, and CURRENT_NODE) provide a convenient means for accessing other CAIS nodes

Many CAIS operations allow the user to omit the relation name when referring to a relationship, defaulting it to "DOT". DOT is therefore referred to as the **default** relation.

The node model also uses the concept of **current process**. This is implicit in all calls to CAIS operations and refers to the currently executing process making the call. It defines the context in which the parameters are to be interpreted. In particular, paths are determined in the context of the current process.

3.1.3 Pathnames

Nodes are accessed by navigating along the relationships. These paths are specified using a pathname syntax. Starting from a given node, a path is followed by traversing a sequence of relationships until the desired node is reached. The **pathname** for this path is made up of the concatenation of the names of the traversed relationships in the same order in which they are encountered.

The syntax of a pathname is a sequence of path elements, each path element representing the traversal of a single relationship. A path element is an apostrophe (" ' ", pronounced "tick") followed by a relation name and a parenthesized relationship key (which may be omitted if the relationship is a unique instance of the relation for this node). If the relation is the default relation DOT, then the path element may be represented simply by a dot (" . ") followed by the key for the default relation DOT. Thus, "'DOT(CONTROLLER)" is the same as ".CONTROLLER".

Pathnames are interpreted relative to a known node. This node may be identified explicitly as an additional argument, the BASE, to many of the CAIS operations. Otherwise, the current process node is used as the starting point for interpretation of the path.

A pathname may begin simply with a relationship key, not prefixed by either " ' " or " . ". This is taken to mean interpretation following the DOT relation of the CURRENT_NODE. Thus "AIRPORT" is the same as " 'CURRENT_NODE.AIRPORT". By convention, the null pathname " " is interpreted as the CURRENT_NODE of the current process.

A pathname may also consist of just a single " . ". This is interpreted as referring to the current process node.

Relation names and relationship keys follow the syntax of Ada identifiers. Upper and lower case are treated as equivalent within such identifiers. For example, all of the following are legal node pathnames, and they would all refer to the same node if the CURRENT_NODE were " 'USER(JONES).TRACKER " and the CURRENT_USER were "JONES":

a. Landing_System'With_unit(Radar)

b. 'User(Jones).TRACKER.Landing_system'with_UNIT(RADAR)

c. 'CURRENT_USER.TRACKER.LANDING_SYSTEM'WITH_unit(radar)

By convention a relationship key of simply "#" is taken to represent the LATEST_KEY (lexicographically last). When creating a node or relationship, use of "#" as the final key of a pathname will cause a key to be automatically assigned, lexicographically following all previous keys for the same relation. This may be used to automatically assign revision identifiers or process keys (see Section 6.2).

The Backus-Naur Form (BNF) for pathnames is given in Table 3-1.

TABLE 3-1
PATHNAME BNF

```
PATHNAME :: = { PATHELEMENT } |
             RELATIONSHIP_KEY { PATHELEMENT } |
             " . "

PATHELEMENT :: = " ' " RELATION_NAME "(" RELATIONSHIP_KEY ")" |
                 " ' " RELATION_NAME |
                 " . " RELATIONSHIP_KEY

RELATION_NAME :: = IDENTIFIER
RELATIONSHIP_KEY :: = IDENTIFIER | " # "
```

3.2 ATTRIBUTES

Both nodes and relationships may have **attributes** which provide information about the node or relationship. Attributes are identified by an attribute name. Each attribute (see Section 3.6 CAIS_ATTRIBUTES) has a list of the values assigned to it, represented using the CAIS_LIST_UTILS (see Section 8.2.2) type called LIST.

Relation names and attribute names both have the same form (that is, the syntax of an Ada identifier), and they must be different from each other for a given node.

This version of the CAIS introduces two pre-defined node attributes: ACCESS_CONTROL and SECURITY_LEVEL.

3.3 GENERAL NODE MANAGEMENT

The operations defined in package CAIS__NODE__MANAGEMENT are applicable to all nodes except where explicitly stated otherwise in the package semantics section.

The creation of nodes for files is performed by the CREATE procedures of the Input/Output packages; the creation of nodes for processes is performed by INVOKE__PROCESS and SPAWN__PROCESS of CAIS__PROCESS__CONTROL (see Section 6.2); the creation of structural nodes is performed by CREATE__NODE (see Section 4.1); the creation of device nodes is performed by the CREATE procedures of CAIS__TERMINAL__SUPPORT (see Section 7.1.1).

To simplify manipulation by Ada programs, an Ada type NODE__TYPE is defined to represent an internal handle for a node. Most procedures either expect a NODE__TYPE parameter, or a pathname, or a combination of a BASE node (specified by a NODE__TYPE parameter) and a pathname relative to it.

3.4 PACKAGE CAIS__NODE__DEFS

This package defines the Ada type NODE__TYPE, which provides an internal (private) reference to CAIS nodes. This is referred to as a node **handle**. It also defines certain enumeration and record types and exceptions useful for node manipulations.

3.4.1 Package Specification

```
with IO__EXCEPTIONS;
package CAIS__NODE__DEFS is

    type NODE__TYPE is      limited private;
    type NODE__KIND is      (FILE, STRUCTURAL, PROCESS, DEVICE);

    subtype NAME__STRING is          STRING;

    subtype NAME__STRING is            STRING;
    subtype FORM__STRING is            STRING;
    subtype RELATIONSHIP__KEY is       STRING;
    subtype RELATION__NAME is          STRING;

    TOP__LEVEL         : constant STRING     := " 'CURRENT__USER";
    CURRENT__NODE      : constant STRING     := " ";
    CURRENT__PROCESS   : constant STRING     := " . ";
    LATEST__KEY        : constant STRING     :=  "#";

    -- Exceptions

    STATUS__ERROR      : exception renames IO__EXCEPTIONS.STATUS__ERROR;
    MODE__ERROR        : exception renames IO__EXCEPTIONS.MODE__ERROR;
    NAME__ERROR        : exception renames IO__EXCEPTIONS.NAME__ERROR;
    USE__ERROR         : exception renames IO__EXCEPTIONS.USE__ERROR;
    LAYOUT__ERROR      : exception renames IO__EXCEPTIONS.LAYOUT__ERROR;

private
    -- implementation-dependent
end CAIS__NODE__DEFS;
```

3.4.2 Package Semantics

```
    TOP_LEVEL          : constant STRING      : = " 'CURRENT_USER";
    CURRENT_NODE       : constant STRING      : = " ";
    CURRENT_PROCESS    : constant STRING      : =  " . ";
    LATEST_KEY         : constant STRING      : =  " # ";
```

Define the standard pathnames for current user's top-level node, current node, current process, and latest key.

```
    STATUS_ERROR  : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR    : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR    : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR     : exception renames IO_EXCEPTIONS.USE_ERROR;
    LAYOUT_ERROR  : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;
```

Renames the corresponding exceptions for the LRM.

3.5 PACKAGE CAIS_NODE_MANAGEMENT

This package defines the general primitives for manipulating, copying, renaming, and deleting nodes and their relationships.

3.5.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_NODE_MANAGEMENT is

    subtype NODE_TYPE          is   CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING        is   CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY   is   CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME      is   CAIS_NODE_DEFS.RELATION_NAME;


    procedure OPEN (NODE:      in out   NODE_TYPE;
                    NAME:      in       NAME_STRING);
    procedure OPEN (NODE:      in out   NODE_TYPE;
                    BASE:      in       NODE_TYPE;
                    KEY:       in       RELATIONSHIP_KEY := "";
                    RELATION:  in       RELATION_NAME := "DOT");

    procedure  CLOSE(NODE: in out NODE_TYPE);

    function  IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;

    function  KIND (NODE: in NODE_TYPE)
              return          CAIS_NODE_DEFS.NODE_KIND;

    function  PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;

    function  PRIMARY_KEY (NODE: in NODE_TYPE)
              return RELATIONSHIP_KEY;

    function  PRIMARY_RELATION (NODE: in NODE_TYPE)
              return RELATION_NAME;
```

```
function  PATH_KEY(NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
function  PATH_RELATION(NODE: in NODE_TYPE) return RELATION_NAME;

procedure GET_PARENT(NODE:   in      NODE_TYPE;
                     PARENT: in out  NODE_TYPE);

procedure COPY_NODE (FROM:        in  NODE_TYPE;
                     TO:          in  NAME_STRING);
procedure COPY_NODE (FROM:        in  NODE_TYPE;
                     TO_BASE:     in  NODE_TYPE;
                     TO_KEY:      in  RELATIONSHIP_KEY := " ";
                     TO_RELATION: in  RELATION_NAME := "DOT");

procedure COPY_TREE (FROM:        in  NODE_TYPE;
                     TO:          in  NAME_STRING);
procedure COPY_TREE (FROM:        in  NODE_TYPE;
                     TO_BASE:     in  NODE_TYPE;
                     TO_KEY:      in  RELATIONSHIP_KEY := " ";
                     TO_RELATION: in  RELATION_NAME := "DOT");

procedure RENAME(NODE:         in  NODE_TYPE;
                 NEW_NAME:     in  NAME_STRING);
procedure RENAME(NODE:         in      NODE_TYPE;
                 NEW_BASE:     in      NODE_TYPE;
                 NEW_KEY:      in      RELATIONSHIP_KEY := " ";
                 NEW_RELATION: in      RELATION_NAME := "DOT");

procedure LINK(TO:         in  NAME_STRING;
               NEW_PATH:   in  NAME_STRING);
procedure LINK(TO_NODE:    in  NODE_TYPE;
               NEW_BASE:   in  NODE_TYPE;
               KEY:        in  RELATIONSHIP_KEY := " ";
               RELATION:   in  RELATION_NAME := "DOT");

procedure UNLINK(NAME:     in  NAME_STRING);
procedure UNLINK(BASE:     in  NODE_TYPE;
                 KEY:      in  RELATIONSHIP_KEY := " ";
                 RELATION: in  RELATION_NAME := "DOT");

procedure DELETE_NODE(NAME:  in      NAME_STRING);
procedure DELETE_NODE(NODE:  in out  NODE_TYPE);

procedure DELETE_TREE(NODE:  in out  NODE_TYPE);

type NODE_ITERATOR is private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN is RELATION_NAME;
subtype NODE_KIND is CAIS_NODE_DEFS.NODE_KIND;

procedure ITERATE(ITERATOR:          out  NODE_ITERATOR;
                  NODE:          in       NODE_TYPE;
                  KIND:          in       NODE_KIND;
                  KEY:           in       RELATIONSHIP_KEY_PATTERN := "*";
                  RELATION:      in       RELATION_NAME_PATTERN := "DOT";
                  PRIMARY_ONLY:  in       BOOLEAN:= TRUE;
```

```
    function MORE (ITERATOR:               in       NODE_ITERATOR)
            return  BOOLEAN;
    procedure GET_NEXT(ITERATOR:           in out   NODE_ITERATOR;
                       NEXT_NODE:          in out   NODE_TYPE);

    procedure SET_CURRENT_NODE(NAME:  in        NAME_STRING);
    procedure SET_CURRENT_NODE(NODE:  in        NODE_TYPE);

    procedure GET_CURRENT_NODE(NODE:  out       NODE_TYPE);

    function IS_SAME(NAME1:  in       NAME_STRING;
                     NAME2:  in       NAME_STRING)
            return BOOLEAN;

    function IS_SAME(NODE1:  in       NODE_TYPE;
                     NODE2:  in       NODE_TYPE)
            return BOOLEAN;

    -- Exceptions

    NAME_ERROR :exception renames CAIS_NODE_DEFS.NAME_ERROR;
    USE_ERROR  :exception renames CAIS_NODE_DEFS.USE_ERROR;


private
    -- implementation-dependent
end CAIS_NODE_MANAGEMENT;
```

3.5.2 *Package Semantics*

```
    subtype NODE_TYPE          is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING        is    CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY   is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME      is    CAIS_NODE_DEFS.RELATION_NAME;
```

The key of a node is the relationship key of the last element of its pathname. Many operations are allowed to take either a pathname or a base-node/key/relation-name.

```
    procedure OPEN (NODE:       in out   NODE_TYPE;
                    NAME:       in       NAME_STRING);
    procedure OPEN (NODE:       in out   NODE_TYPE;
                    BASE:       in       NODE_TYPE;
                    KEY:        in       RELATIONSHIP_KEY := " ";
                    RELATION:   in       RELATION_NAME := "DOT");
```

Returns an open node handle on the designated node. The NAME_ERROR exception will be raised if the node does not exist.

An open node handle acts as if the handle forms a temporary secondary relationship to the node; this means that, if the opened node pointed to is renamed (potentially by another process), the operations on the opened node track the renaming. Tools which require that node relationships remain unchanged between node-level CAIS operations use have the features of the CAIS_NODE_CONTROL package (Section 3.7) to synchronize node usage.

```
    procedure CLOSE(NODE: in out   NODE_TYPE);
```

Severs any association between the node handle and the node and releases any associated lock. This must be done before another OPEN can be done using the same NODE_TYPE variable by the same process.

```
    function IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;
```

Returns TRUE or FALSE according to open status of the node handle.

```
    function KIND (NODE:  in        NODE_TYPE)
            return CAIS_NODE_DEFS.NODE_KIND;
```

Returns the kind of a node, either FILE, PROCESS, STRUCTURAL, or DEVICE.

```
    function PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;
```

Returns the full primary pathname to the node.

```
    function PRIMARY_KEY (NODE:           in        NODE_TYPE)
            return RELATIONSHIP_KEY;
    function PRIMARY_RELATION (NODE:  in        NODE_TYPE)
            return RELATION_NAME;
```

Returns the corresponding part of the last element of the primary path to the node. If the node is a top-level node, the key is the user name, and the relation name is USER.

```
    function PATH_KEY(NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
    function PATH_RELATION(NODE: in NODE_TYPE) return RELATION_NAME;
```

Returns the corresponding part of the last element of the path used to access this node. If the path was an absolute path and this is a top-level node, the relationship key is the user name, and the relation name is USER.

```
    procedure GET_PARENT(NODE:   in       NODE_TYPE;
                         PARENT: in out   NODE_TYPE);
```

Returns the parent node. Generate an exception if NODE is a top-level node.

```
    procedure COPY_NODE (FROM:         in     NODE_TYPE;
                         TO:           in     NAME_STRING);
    procedure COPY_NODE (FROM:         in     NODE_TYPE;
                         TO_BASE:      in     NODE_TYPE;
                         TO_KEY:       in     RELATIONSHIP_KEY := " ";
                         TO_RELATION:  in     RELATION_NAME := "DOT");
```

Copies a node. Any secondary relationships emanating from the original node are recreated in the copy. Unless the target of the original node's relationship is the node itself, then the copied relationship still refers to the same target node. If the target is the node itself, then the copy will have an analogous relationship to itself. It is an error (USE_ERROR) if the node is a process or device node, or if any primary relationships emanate from the original node.

```
    procedure COPY_TREE (FROM:         in     NODE_TYPE;
                         TO:           in     NAME_STRING);
    procedure COPY_TREE (FROM:         in     NODE_TYPE;
                         TO_BASE:      in     NODE_TYPE;
                         TO_KEY:       in     RELATIONSHIP_KEY := " ";
                         TO_RELATION:  in     RELATION_NAME := "DOT");
```

Copies a tree of nodes (formed by primary relationships), as well as their secondary relationships. Secondary relationships between two nodes which are both copied are recreated between the two copies. Secondary relationships emanating from a node which is copied, but which refer to nodes outside the tree being copied, are copied so that they emanate from the copy, but still refer to the old (uncopied) node. The exception USE_ERROR will be raised if any node in the tree is a process or device.

```
procedure RENAME(NODE:              in    NODE_TYPE;
                 NEW_NAME:          in    NAME_STRING);
procedure RENAME(NODE:              in    NODE_TYPE;
                 NEW_BASE:          in    NODE_TYPE;
                 NEW_KEY:           in    RELATIONSHIP_KEY := " ";
                 NEW_RELATION:      in    RELATION_NAME := "DOT");
```

Changes the primary connection to a node and adjusts the PARENT relationship appropriately.

Existing secondary relationships with the renamed node as target will track the renaming. An implementation may raise USE_ERROR if the renaming cannot be accomplished while still maintaining consistent secondary relationships and acircularity of primary relationships. RENAME raises the exception USE_ERROR if a node already exists with the new name.

Existing processes with open node handles track the renamed node; the node's handle acts as if the accessing process had a temporary secondary relationship to the node.

```
procedure LINK (TO:          in    NAME_STRING;
                NEW_PATH:    in    NAME_STRING);
procedure LINK (TO_NODE:     in    NODE_TYPE;
                NEW_BASE:    in    NODE_TYPE;
                KEY:         in    RELATIONSHIP_KEY := " ";
                RELATION:    in    RELATION_NAME := "DOT");
```

Creates a relationship from one existing node to another. This relationship will be identified as a secondary relationship.

The first LINK procedure takes the name of the target node as the TO argument and a NEW_PATH which should lead to it. The base/key/relation are implied by the NEW_PATH. The second LINK procedure takes a handle on the target node, a handle on the NEW_BASE, and an explicit key and relation to be established from NEW_BASE to TO_NODE.

```
procedure UNLINK (NAME:      in    NAME_STRING);
procedure UNLINK (BASE:      in    NODE_TYPE;
                  KEY:       in    RELATIONSHIP_KEY := " ";
                  RELATION:  in    RELATION_NAME := "DOT");
```

Deletes a secondary relationship. Raises USE_ERROR if the specified relationship is a primary relationship or does not exist.

```
procedure DELETE_NODE(NAME:  in       NAME_STRING);
procedure DELETE_NODE(NODE:  in out   NODE_TYPE);
```

Deletes the primary relationship to a node and the node itself. It is an error if any primary relationships emanate from this node.

This delete operation closes NODE, removes the appropriate relationship from the node's parent and updates the node's parent. If a process node is not TERMINATED (see Section 6.1), this action aborts its process. This delete operation cannot be used to delete more than one node in a single operation.

```
procedure DELETE_TREE(NODE:         in out NODE_TYPE);
```

DELETE_TREE deletes a node and recursively deletes all nodes with the designated node as their parent. This operation closes the NODE handle and removes the appropriate relationship from the node's parent. This operation can be used to delete more than one node in a single operation. If DELETE_TREE raises the USE_ERROR exception, no node may be deleted.

```
    type NODE_ITERATOR is private;
    subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
    subtype RELATION_NAME_PATTERN is RELATION_NAME;
    subtype NODE_KIND is CAIS_NODE_DEFS.NODE_KIND;
```

RELATIONSHIP_KEY_PATTERN and RELATION_NAME_PATTERN follow the syntax of relationship keys/relation names, except that a "?" will match any single character and a "*" will match any string of characters.

```
    procedure ITERATE(ITERATOR:              out    NODE_ITERATOR;
                      NODE:               in        NODE_TYPE;
                      KIND:               in        NODE_KIND;
                      KEY:                in        RELATIONSHIP_KEY_PATTERN := "*";
                      RELATION:           in        RELATION_NAME_PATTERN := "DOT";
                      PRIMARY_ONLY:       in        BOOLEAN := TRUE);

    function MORE (ITERATOR:              in        NODE_ITERATOR)
           return  BOOLEAN;

    procedure GET_NEXT(ITERATOR:          in out    NODE_ITERATOR;
                       NEXT_NODE:         in out    NODE_TYPE);
```

These three operations iterate through those nodes referred to from the given NODE, via primary or secondary relationships that have keys/relations satisfying the specified patterns.

The nodes are returned in ASCII lexicographical order by RELATION and then by relationship KEY. The key and relation are available by the functions PATH_KEY and PATH_RELATION (see above). Nodes that are of a different kind than the KIND specified are omitted.

If PRIMARY_ONLY is true, then only primary relationships are considered when creating the iterator. In this case, either PATH_KEY/PATH_RELATION or PRIMARY_KEY/PRIMARY_RELATION may be used to determine the relationship which caused the node to be included in the iteration.

Similarly, these operations iterate through the primary or secondary relationships from the given NODE which have keys/relations satisfying the specified patterns.

```
    procedure SET_CURRENT_NODE(NAME: in    NAME_STRING);
    procedure SET_CURRENT_NODE(NODE: in    NODE_TYPE);
```

Specifies NODE/NAME as the current node.

```
    procedure GET_CURRENT_NODE(NODE: out NODE_TYPE);
```

Opens a handle on the current node. This is equivalent to OPEN(NODE, " 'CURRENT_NODE")

```
    function IS_SAME(NAME1: in    NAME_STRING;
                     NAME2: in    NAME_STRING)
       return BOOLEAN;

    function IS_SAME(NODE1: in    NODE_TYPE;
                     NODE2: in    NODE_TYPE)
       return BOOLEAN;
```

Returns TRUE if both names/node handles refer to the same CAIS node.

3.6 PACKAGE CAIS_ATTRIBUTES

This package supports the definition and manipulation of named attributes for nodes and relationships. Each attribute is a list of the format defined by the package CAIS_LIST_UTILS (see Section 8.2.2). The name of an attribute follows the syntax of an Ada identifier. Upper/lower case distinctions are significant within the value of attributes, but not within the attribute name.

It is anticipated that certain attribute names and their values will be included as part of the CAIS definition. In any case, each implementation must identify those attribute names and values which are reserved or which have special significance.

The operations in this package are overloaded to permit access to nodes and relationships by either the name strings or the node handles. Access by the node handle assures that the operation tracks the node (which may be renamed or locked once open).

3.6.1 Package Specification

```
with CAIS_LIST_UTILS;
with CAIS_NODE_DEFS;
package CAIS_ATTRIBUTES is

     subtype NAME_STRING    is     CAIS_NODE_DEFS.NAME_STRING;
     subtype NODE_TYPE      is     CAIS_NODE_DEFS.NODE_TYPE;
     subtype LIST           is     CAIS_LIST_UTILS.LIST;
     subtype ATTRIB_NAME    is     STRING;
     type FLAG_ENUM         is     (READ_ONLY, INHERIT);

     procedure SET_NODE_ATTRIBUTE(NAME:   in  out NAME_STRING;
                                  ATTRIB: in      ATTRIB_NAME;
                                  VALUE:  in      LIST);
     procedure SET_NODE_ATTRIBUTE(NODE:   in  out NODE_TYPE;
                                  ATTRIB: in      ATTRIB_NAME;
                                  VALUE:  in      LIST);

     procedure SET_PATH_ATTRIBUTE(NAME:   in  out   NAME_STRING;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);
     procedure SET_PATH_ATTRIBUTE(NODE:   in        NODE_TYPE;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);

     procedure GET_NODE_ATTRIBUTE(NAME:   in        NAME_STRING;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);
     procedure GET_NODE_ATTRIBUTE(NODE:   in out    NODE_TYPE;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);

     procedure GET_PATH_ATTRIBUTE(NAME:   in        NAME_STRING;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);
     procedure GET_PATH_ATTRIBUTE(NODE:   in        NODE_TYPE;
                                  ATTRIB: in        ATTRIB_NAME;
                                  VALUE:  in        LIST);

     type ATTRIB_ITERATOR is private;
     subtype ATTRIB_PATTERN is STRING;
```

```
procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:   in out    ATTRIB_ITERATOR;
                                  NAME:       in        NAME_STRING;
                                  PATTERN:    in        ATTRIB_PATTERN := " * ");
procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:   in out    ATTRIB_ITERATOR;
                                  NODE:       in        NODE_TYPE;
                                  PATTERN:    in        ATTRIB_PATTERN := " * ");

procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:   in out    ATTRIB_ITERATOR;
                                  NAME:       in        NAME_STRING;
                                  PATTERN:    in        ATTRIB_PATTERN : = " * ");
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR:    in out    ATTRIB_ITERATOR;
                                  NODE:       in        NODE_TYPE;
                                  PATTERN:    in        ATTRIB_PATTERN := " * ");

function MORE (ITERATOR:        in         ATTRIB_ITERATOR)
        return BOOLEAN;

procedure GET_NEXT(ITERATOR:    in out     ATTRIB_ITERATOR;
                   ATTRIB:         out     ATTRIB_NAME;
                   VALUE:       in out     LIST);

procedure SET_FLAG(NAME:        in         NAME_STRING;
                   ATTRIB:      in         ATTRIB_NAME;
                   WHICH:       in         FLAG_ENUM;
                   TO:          in         BOOLEAN := TRUE);
procedure SET_FLAG(NODE:        in         NODE_TYPE;
                   ATTRIB:      in         ATTRIB_NAME;
                   WHICH:       in         FLAG_ENUM;
                   TO:          in         BOOLEAN : = TRUE);

function FLAG (NAME:     in     NAME_STRING;
               ATTRIB:   in     ATTRIB_NAME;
               WHICH:    in     FLAG_ENUM)
        return BOOLEAN;
function FLAG (NODE:     in     NODE_TYPE;
               ATTRIB:   in     ATTRIB_NAME;
               WHICH:    in     FLAG_ENUM)
        return BOOLEAN;

-- Exceptions

USE_ERROR :exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_ATTRIBUTES;
```

3.6.2 Package Semantics

```
subtype NAME_STRING    is    CAIS_NODE_DEFS.NAME_STRING;
subtype NODE_TYPE      is    CAIS_NODE_DEFS.NODE_TYPE;
subtype LIST           is    CAIS_LIST_UTILS.LIST;
subtype ATTRIB_NAME    is    STRING;
```

Each CAIS node or relationship may have list-valued attributes. They are associated with nodes referred to by a pathname or node handle and with relationships referred to by the last step in a pathname or by the last step associated by a pathname.

```
type FLAG__ENUM  is        (READ__ONLY, INHERIT);
```

The type FLAG__ENUM selects one of two flags associated with each attribute. Attributes with the READ__ONLY flag may not be written. Attributes with no READ__ONLY flag may be read or written. If a node has attributes with the INHERIT flag set, then nodes created with that node as their parent will have the initial values for these attributes copied from those of the parent node.

```
procedure SET__NODE__ATTRIBUTE(NAME:    in out  NAME__STRING;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
procedure SET__NODE__ATTRIBUTE(NODE:    in out  NODE__TYPE;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);

procedure SET__PATH__ATTRIBUTE(NAME:    in out  NAME__STRING;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
procedure SET__PATH__ATTRIBUTE(NODE:    in      NODE__TYPE;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
```

Sets the given node/relationship attribute. If an attribute with the given name already exists, then the existing value is over-written by the given value; if it does not exist, a new attribute is created and set to the given value. Setting the value of the attribute to an empty list deletes the attribute. This operation will fail with USE__ERROR if the attribute is READ__ONLY or if the current process does not have update access to the node.

```
procedure GET__NODE__ATTRIBUTE(NAME:    in      NAME__STRING;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
procedure GET__NODE__ATTRIBUTE(NODE:    in out  NODE__TYPE;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);

procedure GET__PATH__ATTRIBUTE(NAME:    in      NAME__STRING;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
procedure GET__PATH__ATTRIBUTE(NODE:    in      NODE__TYPE;
                               ATTRIB:  in      ATTRIB__NAME;
                               VALUE:   in      LIST);
```

Gets the current value of an attribute. If the attribute has never been set, then these operations return the empty list.

```
type ATTRIB__ITERATOR is private;
subtype ATTRIB__PATTERN is STRING;
```

An attribute iterator is used to sequence through the names of the attributes of a node or a relationship. An ATTRIB__PATTERN has the same syntax as an ATTRIB__NAME, except that "?" stands for any character and "*" stands for zero or more arbitrary characters.

By using simply the pattern "*" it is possible to iterate through the names of all of the non-null attributes of a node.

```
    procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:  in out  ATTRIB_ITERATOR;
                                      NAME:      in      NAME_STRING;
                                      PATTERN:   in      ATTRIB_PATTERN := "*");
    procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:  in out  ATTRIB_ITERATOR;
                                      NODE:      in      NODE_TYPE;
                                      PATTERN:   in      ATTRIB_PATTERN := "*");

    procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:  in out  ATTRIB_ITERATOR;
                                      NAME:      in      NAME_STRING;
                                      PATTERN:   in      ATTRIB_PATTERN:= "*");
    procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:  in out  ATTRIB_ITERATOR;
                                      NODE:      in      NODE_TYPE;
                                      PATTERN:   in      ATTRIB_PATTERN:= "*");

    function MORE (ITERATOR:  in   ATTRIB_ITERATOR)
            return BOOLEAN;

    procedure GET_NEXT(ITERATOR:  in out  ATTRIB_ITERATOR;
                       ATTRIB:       out  ATTRIB_NAME
                       VALUE:     in out  LIST);
```

These operations iterate through the names of the attributes of a node or relationship which match the given pattern. The names are returned in ASCII lexicographical order.

```
    procedure SET_FLAG(NAME:    in   NAME_STRING;
                       ATTRIB:  in   ATTRIB_NAME;
                       WHICH:   in   FLAG_ENUM;
                       TO:      in   BOOLEAN := TRUE);
    procedure SET_FLAG(NODE:    in   NODE_TYPE;
                       ATTRIB:  in   ATTRIB_NAME;
                       WHICH:   in   FLAG_ENUM;
                       TO:      in   BOOLEAN := TRUE);

    function FLAG (NAME:    in    NAME_STRING;
                   ATTRIB:  in    ATTRIB_NAME;
                   WHICH:   in    FLAG_ENUM)
            return BOOLEAN;

    function FLAG (NODE:    in    NODE_TYPE;
                   ATTRIB:  in    ATTRIB_NAME;
                   WHICH:   in    FLAG_ENUM)
            return BOOLEAN;
```

These two operations provide access to the READ_ONLY and INHERIT flags for each attribute. SET_FLAG sets the specified flage. The function FLAG returns the current setting of the flag.

3.7 PACKAGE CAIS_NODE_CONTROL

This version of the CAIS defines only primitives for dynamic access synchronization. Each operation on a node is independent, and both access control and synchronization status are re-checked for each operation. This package defines access synchronization operations at the node levels. For file (and device) nodes, an implementation may define the FORM string to permit an OPEN operation (LRM chapter 14; see also Sections 5 and 7 of this document) which specifies exclusive access; in that case the sequence of file (and device) opening, reading and writing, and closing, is considered a single node-level "operation". Use of file (or device) level access synchronization thus provides for longer transactions at the node level without locking the node's attributes and relationships (only content may be locked by file level OPEN actions). Use of node level access synchronization is intended for control at the level of the node as a whole (content, relationships, and attributes).

3.7.1 Package Specification

```
with CAIS_ATTRIBUTES;
with CAIS_NODE_DEFS;
with CAIS_NODE_CONTROL is

     subtype NODE_TYPE   is CAIS_NODE_DEFS.NODE_TYPE;
     subtype ATTRIB_NAME is CAIS_ATTRIBUTES.ATTRIB_NAME;
     ACCESS_CONTROL  :  constant ATTRIB_NAME  : = "ACCESS_CONTROL";
     SECURITY_LEVEL  :  constant ATTRIB_NAME  : = "SECURITY_LEVEL";

     procedure LOCK (NODE:          in     NODE_TYPE;
                     TIME_LIMIT:    in     DURATION : = DURATION'LAST);

     procedure UNLOCK (NODE:        in     NODE_TYPE);

private
     -- implementation-dependent
end CAIS_NODE_CONTROL;
```

3.7.2 Package Semantics

```
     subtype NODE_TYPE   is CAIS_NODE_DEFS.NODE_TYPE;
     subtype ATTRIB_NAME is CAIS_ATTRIBUTES.ATTRIB_NAME;
     ACCESS_CONTROL  :  constant ATTRIB_NAME  : = "ACCESS_CONTROL";
     SECURITY_LEVEL  :  constant ATTRIB_NAME  : = "SECURITY_LEVEL";
```

The CAIS provides two predefined attribute names for acces control: ACCESS_CONTROL for discretionary access control and SECURITY_LEVEL for non-discretionary access control. These attributes may be set at node creation (by inclusion in the FORM string — see Section 4.1) or later with SET_NODE_ATTRIBUTE (see Section 3.6).

```
     procedure LOCK (NODE:          in     NODE_TYPE;
                     TIME_LIMIT:    in     DURATION : = DURATION'LAST);

     procedure UNLOCK (NODE:        in     NODE_TYPE);
```

Locks/unlocks the designated node for a series of updates. When a node is locked, any other process that attempts to modify any attribute, relationship, or content of the node will receive an exception. If the node is already locked, then LOCK will be delayed until the node is unlocked or until the time limit expires. In the later case an exception will be raised.

3.8 PRAGMATICS

Several private types are defined as part of the CAIS Node Model. The actual implementation of these types may vary from one CAIS implementation to the next. Nevertheless, it is important to establish certain minimums for each type to enhance portability.

a.	NAME_STRING	At least 255 characters in a CAIS pathname.
b.	RELATIONSHIP_KEY KEY_STRING	At least 20 characters must be significant in (relationship) key.

c.	ATTRIB_NAME RELATION_NAME	At least 20 characters must be significant in attribute/relation names.
d.	Tree-height	At least 10 levels of heirarchy must be supported for the primary relationships.
e.	Record size number	At least 32767 bits per record must be supported.
f.	Open node count	Each process must be able to have at least 15 nodes open simultaneously.

4. CAIS STRUCTURAL NODES

Structural nodes are special nodes in the sense that they do not contain contents as do the other nodes of the CAIS model. Their purpose is solely to be carriers of common information about other nodes related to the structural node. Structural nodes are typically used to create conventional directories, configuration objects, etc.

The package CAIS_STRUCTURAL_NODES defines the primitive operations for creating structural nodes. All other operations for structural nodes are defined in Section 3.

4.1 PACKAGE CAIS_STRUCTURAL_NODES

4.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_STRUCTURAL_NODES is

     subtype NODE_TYPE              is      CAIS_NODE_DEFS.NODE_TYPE;
     subtype NAME_STRING            is      CAIS_NODE_DEFS.NAME_STRING;
     subtype FORM_STRING            is      CAIS_NODE_DEFS.FORMS_STRING;
     subtype RELATIONSHIP_KEY       is      CAIS_NODE_DEFS.RELATIONSHIP_KEY;
     subtype RELATION_NAME          is      CAIS_NODE_DEFS.RELATION_NAME;

     procedure CREATE_NODE(NAME:            in      NAME_STRING;
                           FORM:            in      FORM_STRING := " ");
     procedure CREATE_NODE(BASE:            in      NODE_TYPE;
                           KEY:             in      RELATIONSHIP_KEY := " ";
                           RELATION:        in      RELATION_NAME := "DOT";
                           FORM:            in      FORM_STRING := " ");
     procedure CREATE_NODE(NODE:            in out     NODE_TYPE;
                           NAME:            in         NAME_STRING;
                           FORM:            in         FORM_STRING := " ");
     procedure CREATE_NODE(NODE:            in out     NODE_TYPE;
                           BASE:            in         NODE_TYPE;
                           KEY:             in         RELATIONSHIP_KEY := " ";
                           RELATION:        in         RELATION_NAME := "DOT";
                           FORM:            in         FORM_STRING := " ");

private
     -- implementation-dependent
end CAIS_STRUCTURAL_NODES;
```

4.1.2 Package Semantics

```
     subtype NODE_TYPE              is      CAIS_NODE_DEFS.NODE_TYPE;
     subtype NAME_STRING            is      CAIS_NODE_DEFS.NAME_STRING;
     subtype FORM_STRING            is      CAIS_NODE_DEFS.FORM_STRING;
     subtype RELATIONSHIP_KEY       is      CAIS_NODE_DEFS.RELATIONSHIP_KEY;
     subtype RELATION_NAME          is      CAIS_NODE_DEFS.RELATION_NAME;
```

```
procedure CREATE_NODE(NAME:      in      NAME_STRING;
                      FORM:      in      FORM_STRING := " ");
procedure CREATE_NODE(BASE:      in      NODE_TYPE;
                      KEY:       in      RELATIONSHIP_KEY := " ";
                      RELATION:  in      RELATION_NAME := "DOT";
                      FORM:      in      FORM_STRING := " ");
procedure CREATE_NODE(NODE:      in out  NODE_TYPE;
                      NAME:      in      NAME_STRING;
                      FORM:      in      FORM_STRING := " ");
procedure CREATE_NODE(NODE:      in out  NODE_TYPE;
                      BASE:      in      NODE_TYPE;
                      KEY:       in      RELATIONSHIP_KEY := " ";
                      RELATION:  in      RELATION_NAME := "DOT";
                      FORM:      in      FORM_STRING := " ");
```

Creates a structural node with its primary relationship and parent node implied by the NAME in the first and third procedures and given explicitly in the second and fourth procedures.

The last two procedures return a node handle allowing immediate access to attributes and relationships.

If non-null, the FORM parameter provides initial values for attributes of the node, using Ada aggregate syntax, with each attribute name followed by a right-arrow (" = > ") and the attribute value (see Section 8.2.2 CAIS_LIST_UTILS for the syntax of attribute value).

5. CAIS FILE NODES

CAIS file nodes are nodes that represent information about and contain external files. The underlying model for the content of such a node is that of a file of data items, accessible randomly by some index or indices or sequentially. The basic operations on such files are provided by the Ada packages for Input/Output specified in Chapter 14 of the Ada LRM. While the semantics of the packages as specified in the LRM are fully adhered to, the CAIS imposes additional requirements on those semantics that the LRM designates as being implementation-defined. These requirements ensure consistent cooperation between the file-related, node-related, and device-related operations.

The CAIS defines additional Input/Output packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, CAIS_TEXT_IO, and CAIS_INTERACTIVE_IO. The first three packages are identical to the Input/Output packages specified in the Ada LRM, except that additional subprograms are added supporting more convenient and efficient file management operations by exploiting the CAIS Node Model. The package CAIS_INTERACTIVE_IO defines additional Input/Output facilities appropriate fcr files which are assigned to terminals.

To insure the consistency of file- and node-related operations the CAIS imposes the following two constraints on all I/O packages:

> A file must first be made accessible to an Ada program by an OPEN or CREATE, specifying the external file by a NAME and a FORM, both character strings. The formats of these strings are not specified in the Ada LRM. The CAIS requires the formats and semantics for NAME and FORM to adhere to the specifications given in Sections 3 and 4, respectively. Thus file names have the syntax of node pathnames.
>
> The CREATE operations both establish a new external file (as described in Chapter 14 of the Ada LRM) and have the side effect of creating the node for the file. The file node's primary relationship and parent node are implied by the NAME parameter. The DELETE operations have the side effect of deleting the node itself. DELETE operations are not legal if a file's node has primary relationships emanting from it. I/O DELETE operations require that the file be open; CAIS_NODE_MANAGEMENT DELETE operations require only that the node be open (but they also delete the contents with the deletion of the node itself).

While an implementation may provide a mechanism for file creation and opening to specify access synchronization, via the FORM parameter, that access synchronization refers to the file contents level only. To utilize node level access synchronization, the user must open the node explicitly and specify node synchronization operations (see Section 3.7). Files may be opened with or without node handles being opened, and nodes may be open before or while associated file handles are open.

5.1 Ada LRM INPUT/OUTPUT

5.1.1 Package IO_EXCEPTIONS

This package is specified by Chapter 14 of the Ada LRM. The LRM-defined package provides the definition for all exceptions generated by the input/output packages.

5.1.2 Package SEQUENTIAL_IO

This package provides sequential access to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of additional pragmatic requirements it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.1.3 Package DIRECT_IO

This package provides for direct-access input/output to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of pragmatic and additional implied semantic requirements, it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.1.4 Package TEXT_IO

This package provides sequential formatted input/output to ASCII text files. This package is specified in Chapter 14 of the Ada LRM, however, because of pragmatics and additional implied semantics, it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.2 CAIS INPUT/OUTPUT

5.2.1 CAIS File Management

Section 14.2.1 of the Ada LRM defines the file management operations CREATE and OPEN that are included in each of the Ada LRM Input/Output packages. These operations use a pathname as identification of the external file. In the CAIS model, this pathname implies a navigation along relationships to reach the node whose content represents the desired external file.

In the CAIS, the navigation operations of CAIS_NODE_MANAGEMENT allow the identification of the node associated with a file by means of a pathname and also by means of an opened node handle, or a base node and a relationship identification (i.e., relation name and relationship key) leading to the desired node.

The procedures and functions described in this section provide for the control of external files; their declarations are repeated in each of the three packages for CAIS sequential, direct, and text input/output. In order to provide for a smooth transition from a file node to the file itself, and to prevent unnecessary repetitions of navigations, the file management operations CREATE and OPEN included in the packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, and CAIS_TEXT_IO are provided in overloaded versions:

```
    subtype NODE_TYPE is CAIS_NODE_DEFS.NODE_TYPE;
    procedure CREATE (FILE:         in out    FILE_TYPE;
                      MODE:         in        FILE_MODE;
                      BASE:         in        NODE_TYPE;
                      KEY:          in        RELATIONSHIP_KEY: = " ";
                      RELATION:     in        RELATION_NAME:  = "DOT";
                      FORM:         in        FORM_STRING: = " ");

    procedure OPEN (FILE:           in out    FILE_TYPE;
                    MODE:           in        FILE_MODE;
                    BASE:           in        NODE_TYPE;
                    KEY:            in        RELATIONSHIP_KEY: = " ";
                    RELATION:       in        RELATION_NAME:  = "DOT";
                    FORM:           in        FORM_STRING: = " ");
    procedure OPEN (FILE:           in out    FILE_TYPE;
                    MODE:           in        FILE_MODE;
                    NODE:           in        NODE_TYPE;
                    FORM:           in        FORM_STRING: = " ");
```

The semantics of the operations are the same as specified in the Ada LRM Section 14.2.1 and CAIS Section 5.0, except that the external file is identified by means of the associated node handle or BASE, KEY, RELATION.

In addition, the following operation is provided to obtain an opened node handle for the node associ[illegible] with a file:

```
procedure OPEN__NODE(NODE:  in out    NODE__TYPE;
                     FILE:  in        FILE__TYPE);
```

The exception STATUS__ERROR is raised if either the actual parameter for FILE is a closed file handle or the actual parameter for NODE is an already open node handle.

5.2.2 Package CAIS__SEQUENTIAL IO

This package provides sequential access to files/devices. This package is specified by Chapter 14 of the Ada RML; however, because of additional pragmatic requirements it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations given in Section 5.2.1 are added to the package.

5.2.3 Package CAIS__DIRECT__IO

This package provides for direct-access input/output to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of pragmatic and additional implied semantic requirements, it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations in Section 5.2.1 are added to the package.

A conforming implementation should support access with package CAIS__SEQUENTIAL__IO to an external file created and/or maintained with CAIS__DIRECT__IO. (This requires that the generic instantiations of both packages utilize the identical ELEMENT__TYPE.)

5.2.4 Package CAIS__TEXT__IO

This package provides sequential formatted input/output to ASCII text files. This package is specified in Chapter 14 of *the Ada LRM; however,* because of pragmatics and additional implied semantics, it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations given in Section 5.2.1 are added to the package.

A conforming implementation that supports CAIS__INTERACTIVE__IO provides additional semantics in the CAIS__TEXT__IO package for the CAIS__TEXT__IO procedures and functions which are used in reference to printer-type terminals and Video Display Terminal (VDT) type terminals associated with an object of type CAIS__INTERFACE__IO.INTERACTIVE__TERMINAL.

The line terminator, page terminator, and file terminator characters are implementation-dependent.

A VDT functions identically to a hardcopy terminal unless bounds are set for the line length and/or page length. For a cursor-addressable VDT, the current column number and current line number of the associated input file and output file indicate the column number and line number, respectively, on the VDT display. The character position in the upper left corner of the VDT display is the first column of the first line of the first page.

The following procedures have additional semantics when used in reference to a terminal.

```
procedure SET__LINE__LENGTH(FILE :   in    FILE__TYPE; TO : in COUNT);
procedure SET__LINE__LENGTH(TO :     in     COUNT);
```

The exception USE__ERROR is raised if the value of TO is greater than the number of character positions on a line of the display.

```
procedure SET__PAGE__LENGTH(FILE :   in     FILE__TYPE; TO : in COUNT);
procedure SET__PAGE__LENGTH(TO :     in     COUNT);
```

In reference to a VDT the exception USE__ERROR is raised if the value of TO is greater than the number of lines on the display.

```
procedure NEW_LINE(FILE :          in    FILE_TYPE;
                   SPACING :       in    POSITIVE_COUNT := 1);
procedure NEW_LINE(SPACING :       in    POSITIVE_COUNT := 1);
```

In reference to a VDT the active position is moved to the first column of the line below the current line. If the active position was on the last line of the page, NEW_LINE causes all lines of the display to be moved upward such that the top line(s) is lost and the last line of the page is blank.

SPACING acts as defined in the LRM.

```
procedure NEW_PAGE(FILE :   in        FILE_TYPE);
procedure NEW_PAGE;
```

In reference to a VDT the screen is cleared and the active position is moved to the first column of the first line of the display.

```
procedure GET(. . .);
```

In reference to a cursor-addressable VDT with a bounded line length the GET procedures clear a portion of the display starting at the active position and equal in length to the maximum possible length of the item to be read. The active position is not changed. The data to be read is buffered as the user enters it. Implementation defined editing operations are permitted. No characters other than the printable characters and horizontal tab (HT) may be returned.

```
procedure SET_ERROR (FILE :   in        FILE_TYPE);
```

Provides an open file handle to be used for current error output. The exception MODE_ERROR is raised if the mode of FILE is IN_FILE.

```
function STANDARD_ERROR return FILE_TYPE;
```

Returns error output set at start of program execution.

```
function CURRENT_ERROR return FILE_TYPE;
```

Returns current error output, set by SET_ERROR.

5.2.5 Package CAIS_INTERACTIVE_IO

This package defines input and output facilities appropriate to files which are assigned to terminals.

The package provides for association of input and output text files with an output logging file. It also provides for turning on and off local echoing of input, association of a prompt string with terminal input, and simplistic random access within a terminal display.

Finally, this package defines a standard error-output text file which is used for error messages which are generated during program execution, but which would be missed if they were output to a re-directed standard output.

5.2.5.1 Package Specification

```
with CAIS_TEXT_IO;
with CAIS_NODE_DEFS;
package CAIS_INTERACTIVE_IO is

    subtype FILE_TYPE is CAIS_TEXT_IO.FILE_TYPE;

    type INTERACTIVE_TERMINAL is limited private;
    procedure ASSOCIATE (TERMINAL :   in out   INTERACTIVE_TERMINAL;
                         INFILE :     in       FILE_TYPE;
                         OUTFILE :    in       FILE_TYPE);

    procedure SET_LOG (TERMINAL :     in out   INTERACTIVE_TERMINAL;
                       LOG_FILE :     in       FILE_TYPE);

    function LOG (TERMINAL : in INTERACTIVE_TERMINAL)
            return FILE_TYPE;

    type CURSOR_POSITION is
        record
            LINE : POSITIVE;
            COLUMN : POSITIVE:
        end record;

    procedure SET_CURSOR (TERMINAL :   in out   INTERACTIVE_TERMINAL;
                          POSITION :   in       CURSOR_POSITION);

    function CURSOR (TERMINAL : in out INTERACTIVE_TERMINAL)
        return CURSOR_POSITION;

    function SIZE (TERMINAL : in out INTERACTIVE_TERMINAL)
        return CURSOR_POSITION;

    procedure UPDATE (TERMINAL : in out INTERACTIVE_TERMINAL);

    procedure SET_ECHO (TERMINAL :     in out   INTERACTIVE_TERMINAL;
                        TO :           in       BOOLEAN := TRUE);

    function ECHO (TERMINAL : in INTERACTIVE_TERMINAL) return BOOLEAN;

    procedure SET_PROMPT (TERMINAL :   in    INTERACTIVE_TERMINAL;
                          TO :         in    STRING);

    function PROMPT (TERMINAL : in INTERACTIVE_TERMINAL) return STRING;

    -- Exceptions

    LAYOUT_ERROR    :    exception renames CAIS_NODE_DEFS.LAYOUT_ERROR;
    MODE_ERROR      :    exception renames CAIS_NODE_DEFS.MODE_ERROR;
    STATUS_ERROR    :    exception renames CAIS_NODE_DEFS.STATUS_ERROR;
    USE_ERROR       :    exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_INTERACTIVE_IO;
```

5.2.5.2 Package Semantics

```
    procedure ASSOCIATE (TERMINAL :   in out     INTERACTIVE_TERMINAL;
                         INFILE :     in         FILE_TYPE;
                         OUTFILE :    in         FILE_TYPE);
```

Associates the INFILE (a file of mode IN_FILE) and the file OUTFILE (a file of mode OUT_FILE) with the TERMINAL. The exception MODE_ERROR is raised if the mode of INFILE is OUT_FILE or the mode of OUTFILE is IN_FILE. The exception STATUS_ERROR is raised if either INFILE or OUTFILE is not open.

```
procedure SET_LOG (TERMINAL :     in out     INTERACTIVE_TERMINAL;
                   LOG_FILE :     in         FILE_TYPE);
```

Sets LOG_FILE as the file on which the output log is written. When logging is active, all output is simultaneously provided to both the output file and the log file. Logging associations on the standard input and standard output text files are required to be preserved across program invocations. The exception MODE_ERROR is raised if the mode of LOG_FILE is IN_FILE. The exception STATUS_ERROR is raised if CAIS_TEXT_IO.IS_OPEN(LOG_FILE) returns FALSE.

```
function LOG (TERMINAL :      in          INTERACTIVE_TERMINAL)
                                  return  FILE_TYPE;
```

Returns the current logging file associated with TERMINAL. The file handle returned is not open if not logging.

```
type CURSOR_POSITION is
    record
        LINE : POSITIVE;
        COLUMN : POSITIVE;
    end record;
```

CURSOR_POSITION identifies the line and column numbers of a terminal.

```
procedure SET_CURSOR (TERMINAL :  in out  INTERACTIVE_TERMINAL;
                      POSITION :  in      CURSOR_POSITION);
```

Moves the active position on the display to that specified by POSITION. The exception LAYOUT_ERROR is raised if the LINE or COLUMN number exceeds PAGE_LENGTH or LINE_LENGTH, respectively, when bounded. For a hardcopy terminal the exception USE_ERROR is raised if the LINE or COLUMN number is less than the current line or column number, respectively.

```
function CURSOR (TERMINAL :   in out    INTERACTIVE_TERMINAL)
                               return   CURSOR_POSITION;
```

Returns the current CURSOR_POSITION.

```
function SIZE (TERMINAL :     in out    INTERACTIVE_TERMINAL)
                               return   CURSOR_POSITION;
```

Returns the number of lines and number of columns on the terminal.

```
procedure UPDATE (TERMINAL :  in out    INTERACTIVE_TERMINAL);
```

Forces all data that has not already been output to the physical terminal to be output immediately.

```
procedure SET_ECHO (TERMINAL :  in out  INTERACTIVE_TERMINAL;
                    TO :        in      BOOLEAN := TRUE);
```

Turns on (TRUE) or off (FALSE) echoing for input file.

```
function ECHO (TERMINAL : in INTERACTIVE_TERMINAL) return BOOLEAN;
```

Indicates current state of echoing.

```
procedure SET_PROMPT (TERMINAL in     INTERACTIVE_TERMINAL;
                      TO :     in     STRING);
```

Sets prompting string for TERMINAL. All future requests for a line of input from TERMINAL will output prompt string first. The prompting string and any echoed input are also copied to the log file, if any.

function PROMPT (TERMINAL : in INTERACTIVE_TERMINAL) return STRING;

Returns current prompt string for input file.

5.3 PRAGMATICS

a.	DIRECT_IO CAIS_DIRECT_IO	Each element of a *direct-access file* is *selected* by an integer index of type COUNT. A conforming implementation must at least support a range of indices from one to 32767 (2**15-1).
b.	SEQUENTIAL_IO CAIS_SEQUENTIAL_IO DIRECT_IO CAIS_DIRECT_IO	A conforming implementation must support generic instantiation of these packages with any (non-limited) constrained Ada type whose maximum size in bits (as defined by the attribute ELEMENT_TYPE'SIZE) is at least 32767. A conforming implementation must also support instantiation with unconstrained record types which have default constraints and a maximum size in bits of at least 32767, and may (but need not) use variable length elements to conserve space in the external file.
c.	TEXT_IO CAIS_TEXT_IO	A conforming implementation must support files with at least 32767 records/lines in total and at least 32767 lines per page. A conforming implementation must support at least 255 columns per line.

6. CAIS_PROCESS_NODES

Each time an Ada program is invoked, a process node is created to represent the execution of the program. Even where the Ada program uses tasking, the execution of the program and its tasks is treated as a single CAIS **process**. This use of the term process does not preclude the CAIS implementation from devoting more than one host process or one physical processor to the execution of the single process.

The mechanism by which a user enters the APSE (e.g., logs on) is not defined as part of the CAIS. The facility to verify access rights to a system via user ID and password, for example, and to establish privileges and resource rights and quotas may be supported either by the APSE or its underlying implementation.

Each time a user enters the APSE a root process node is created dynamically at the top-level node of the user. This root process node initiates a tree of dependent processes. The primary relationship for the node of the root process emanates from the top-level node of the user. It has relation name "JOB" and a relationship key assigned by the APSE or underlying implementation of the APSE. This key is unique for each process node created by the user. In other words, the format 'USER (XXX)' JOB (YYY) is the absolute pathname of a job.

The root process node exists for the duration of the job's existence in the APSE. When the user's job terminates, the root process is terminated and the root process node is deleted.

A process may create other processes by invocation. This act of invocation creates both the node representing the process and the process itself. The new process is a child of the invoking process. The primary relationship of the nodes of these processes emanates from the invoking process with relation name "DOT" and a relationship key that is unique among nodes bearing the DOT relation with the invoker. The relationship key is an identifier assigned by the invoking process. By default, the 'CURRENT_NODE relationship of the new process is established to be the 'CURRENT_NODE of the invoking process.

A process is identified by providing a pathname to its process node (see CAIS Node Model, Section 3). List-valued attributes and secondary relationships for a process are established using the general node manipulation routines (see CAIS Node Model, Section 3).

Processes may communicate with each other using the techniques and procedures described in CAIS_PROCESS_COMMUNICATION (see Section 6.3). The basic capability provides for sending and receiving messages over channels between processes, using a queueing model.

Processes may interrupt each other using the techniques and procedures described in CAIS_PROCESS_INTERRUPTS (see Section 6.5). This basic capability allows for signalling and responding to "pseudo-interrupts," using an asynchronous model for the delivery of the signal. The response to any pseudo-interrupt is definable by the Ada program before the delivery of the signal.

6.1 PACKAGE CAIS_PROCESS_DEFS

This package defines types and exceptions associated with process nodes.

6.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_PROCESS_DEFS is

    type PROCESS_STATUS is
        (READY, SUSPENDED, ABORTING, TERMINATING);
    type COMPLETION_STATUS is (ABORTED, TERMINATED);

    ROOT_PROCESS:       constant STRING := " 'CURRENT_JOB";
    CURRENT_PROCESS: constant STRING := "";

     -- Exceptions

    NAME_ERROR:          exception renames CAIS_NODE_DEF.NAME_ERROR;
    USE_ERROR:           exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_DEFS;
```

6.1.2 Package Semantics

```
    type PROCESS_STATUS is
        (READY, SUSPENDED, ABORTING, TERMINATING);
```

The PROCESS_STATUS is the state a process is in when viewed from another process. Table 6-1 indicates the states and the events which will cause transition from one state to another. In the READY state a process is actually running or is waiting for resources.

TABLE 6-1
PROCESS STATE TABLE

STATE / OPERATION	READY	SUSPENDED	ABORTING	TERMINATING
TERMINATE	TERMINATING	TERMINATING	—	—
ABORT	ABORTING	ABORTING	—	ABORTING
SUSPEND	SUSPENDED	—	N/A	N/A
RESUME	—	READY	N/A	N/A

N/A: marks events that are not applicable to the state specified.
——: marks events that have no effect on the state.

Transition to a state as the result of an event is instantaneous with the occurrence of the event. As the state-transition diagram indicates, there is no transition from an ABORTING or TERMINATING state into any running state.

```
    type COMPLETION_STATUS is (ABORTED, TERMINATED);
```

COMPLETION_STATUS is made available to an invoking process upon completion of a descendant process. These are representative states of a process, since at the time of their receipt the process may have already ceased to exist, depending upon the mechanism provided in the implementation underlying the CAIS for handling completed processes.

```
    ROOT_PROCESS:         constant STRING: = " 'CURRENT_JOB";
    CURRENT_PROCESS:   constant STRING: = " . ";
```

ROOT_PROCESS and CURRENT_PROCESS are two strings defined to represent respectively the root process of the current job and the current process.

6.2 PACKAGE CAIS_PROCESS_CONTROL

This package provides support for the invocation of a program. A program can be invoked using the synchronous model, in which the calling task is suspended during the life of the dependent process and is resumed when the dependent process terminates, either normally or abnormally. A program can also be spawned using an asynchronous model, in which the calling task continues execution after the call which creates a dependent process.

6.2.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_PROCESS_DEFS;
with CAIS_TEXT_IO;
with CAIS_TEXT_UTILS;
package CAIS_PROCESS_CONTROL is

    subtype PROGRAM_STRING      is    STRING;
    subtype RESULTS_STRING      is    CAIS_TEXT_UTILS.TEXT;
    subtype PARAMS_STRING       is    CAIS_TEXT_UTILS.TEXT;
    subtype NAME_STRING         is    CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY    is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype COMPLETION_STATUS   is    CAIS_PROCESS_DEFS.COMPLETION_STATUS;
    subtype FILE_TYPE           is    CAIS_TEXT_IO.FILE_TYPE;
    subtype NODE_TYPE           is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype PROCESS_STATUS      is    CAIS_PROCESS_DEFS.PROCESS_STATUS;

    UNIQUE_CHILD_KEY: STRING renames CAIS_NODE_DEFS.LATEST_KEY;

    procedure INVOKE_PROCESS (PROGRAM:    in       PROGRAM_STRING;
                              PARAMS:     in       PARAMS_STRING;
                              RESULTS:    in out   RESULTS_STRING;
                              STATUS:        out   COMPLETION_STATUS;
                              KEY:        in       RELATIONSHIP_KEY:= UNIQUE_CHILD_KEY;
                              STD_IN:     in       FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_INPUT;
                              STD_OUT:    in       FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_OUTPUT;
                              STD_ERR:    in       FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_ERROR;
                              CURR_NODE:  in       NAME_STRING :=
                                          " 'CURRENT_NODE");

    procedure SPAWN_PROCESS (PROGRAM:    in       PROGRAM_STRING;
                             PARAMS:     in       PARAMS_STRING;
                             NODE:       in out   NODE_TYPE;
                             KEY:        in       RELATIONSHIP_KEY:=
                                         UNIQUE_CHILD_KEY;
                             STD_IN:     in       FILE_TYPE :=
                                         CAIS_TEXT_IO.CURRENT_INPUT;
                             STD_OUT:    in       FILE_TYPE :=
                                         CAIS_TEXT_IO.CURRENT_OUTPUT;
                             STD_ERR:    in       FILE_TYPE :=
                                         CAIS_TEXT_IO.CURRENT_ERROR;
                             CURR_NODE:  in       NAME_STRING :=
                                         " 'CURRENT_NODE");
    procedure AWAIT_PROCESS (PROCESS:       in out  NODE_TYPE;
                             RESULTS:       in out  RESULTS_STRING;
                             STATUS:           out  COMPLETION_STATUS;
                             LIMIT:         in      DURATION := DURATION' LAST);

    procedure GET_PARAMS (PARAMS:           in out  PARAMS_STRING);

    procedure RETURN_TERMINATED(RESULTS:    in      RESULTS_STRING;

    procedure RETURN_ABORTED(RESULTS:       in      RESULTS_STRING);

    procedure ABORT_PROCESS (PROCESS:       in      NAME_STRING);
    procedure ABORT_PROCESS (NODE:          in      NODE_TYPE);

    procedure SUSPEND_PROCESS(PROCESS:      in      NAME_STRING);
    procedure SUSPEND_PROCESS(NODE:         in      NODE_TYPE);

    procedure RESUME_PROCESS (PROCESS:      in      NAME_STRING);
    procedure RESUME_PROCESS (NODE:         in      NODE_TYPE);

    function STATE_OF_PROCESS (PROCESS:     in      NAME_STRING) return PROCESS_STATUS;
    function STATE_OF_PROCESS (NODE:        in      NODE_TYPE) return PROCESS_STATUS;

    function JOB_INPUT return FILE_TYPE;
    function JOB_OUTPUT return FILE_TYPE;

    -- Exceptions

    USE_ERROR: exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_CONTROL;
```

6.2.2 Package Semantics

```
    subtype PROGRAM_STRING      is    STRING;
    subtype RESULTS_STRING      is    CAIS_TEXT_UTILS.TEXT;
    subtype PARAMS_STRING       is    CAIS_TEXT_UTILS.TEXT;
    subtype NAME_STRING         is    CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY    is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype COMPLETION_STATUS   is    CAIS_PROCESS_DEFS.COMPLETION_STATUS;
    subtype FILE_TYPE           is    CAIS_TEXT_IO.FILE_TYPE;
    subtype NODE_TYPE           is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype PROCESS_STATUS      is    CAIS_PROCESS_DEFS.PROCESS_STATUS;

    UNIQUE_CHILD_KEY: STRING renames CAIS_NODE_DEFS.LATEST_KEY;

    procedure INVOKE_PROCESS (PROGRAM:    in      PROGRAM_STRING;
                              PARAMS:     in      PARAMS_STRING;
                              RESULTS:    in out  RESULTS_STRING;
                              STATUS:        out  COMPLETION_STATUS;
                              KEY:        in      RELATIONSHIP_KEY:=
                                                  UNIQUE_CHILD_KEY;
                              STD_IN:     in      FILE_TYPE :=
                                                  CAIS_TEXT_IO.CURRENT_INPUT;
                              STD_OUT:    in      FILE_TYPE :=
                                                  CAIS_TEXT_IO.CURRENT_OUTPUT;
```

```
                         STD_ERR:    in     FILE_TYPE :=
                                            CAIS_TEXT_IO.CURRENT_ERROR;
                         CURR_NODE: in      NAME_STRING := " 'CURRENT_NODE");
```

Creates a new node and a new process and passes a list of parameters to the new process. The calling task can either supply the KEY or the CAIS implementation will assign a unique key via *UNIQUE_CHILD_KEY*. The calling task is suspended until the new process terminates or aborts. The results are returned as a list, along with an enumeration specifying the process's completion status. The node of the terminated process is automatically deleted upon termination.

```
    procedure SPAWN_PROCESS (PROGRAM:      in       PROGRAM_STRING;
                            PARAMS:        in       PARAMS_STRING;
                            NODE:          in out   NODE_TYPE;
                            KEY:           in       RELATIONSHIP_KEY: =
                                                    UNIQUE_CHILD_KEY;
                            STD_IN:        in       FILE_TYPE :=
                                                    CAIS_TEXT_IO.CURRENT_INPUT;
                            STD_OUT:       in       FILE_TYPE :=
                                                    CAIS_TEXT_IO.CURRENT_OUTPUT;
                            STD_ERR:       in       FILE_TYPE :=
                                                    CAIS_TEXT_IO.CURRENT_ERROR;
                            CURR_NODE:     in       NAME_STRING :=
                                                    " 'CURRENT_NODE");
```

Results in a new node and a new process being created to represent the execution of the specified program. Control returns to the invoking process. This invocation provides no technique for coordination of the new process with its parent, except that termination of the parent will not be completed until all children are terminated or aborted. Similarly, no technique is provided for returning a result string to the invoking process. Communication between parent and child can be provided using the techniques provided in CAIS_PROCESS_COMMUNICATION.

```
    procedure AWAIT_PROCESS (PROCESS: in out  NODE_TYPE;
                            RESULTS: in out  RESULTS_STRING;
                            STATUS:      out COMPLETION_STATUS;
                            LIMIT:    in      DURATION : =DURATION'LAST);
```

Suspend the calling task and wait for the process created by SPAWN_PROCESS to complete. The USE_ERROR exception is generated if this is not the first attempt to wait for this descendant process. The result parameter and COMPLETION_STATUS are provided by spawned process's return, even if the process completes execution before the call is made. A time limit is provided in which the parameters must be received or a TIME_OUT exception is raised.

```
    procedure GET_PARAMS(PARAMS: in out   PARAMS_STRING);
```

Retrieve the parameters passed to a process by its caller.

```
    procedure RETURN_TERMINATED (RESULTS: in       RESULTS_STRING);
```

Await termination of all descendant processess, and then return the specified result parameter to the calling process. The COMPLETION_STATUS will be TERMINATED.

```
    procedure RETURN_ABORTED (RESULTS: in       RESULTS_STRING);
```

Abort the *current process* (and all of its descendant processes) and then return the specified result parameter to the calling process. The COMPLETION_STATUS will be ABORTED.

```
procedure ABORT_PROCESS (PROCESS: in        NAME_STRING);
procedure ABORT_PROCESS (NODE:    in        NODE_TYPE);
```

Aborts the specified process and recursively forces any descendants of the named process to be aborted. The sequencing of the process abortions is not specified. ABORT_PROCESS returns control to the issuing process immediately. At that time, if the state of the aborted process is examined, it will be either ABORTING or the process will be non-existent. This node associated with the aborted process remains until explicitly deleted by the invoking process.

The COMPLETION_STATUS of the process will be ABORTED. ABORT_PROCESS can be used by a process to abort itself.

```
procedure SUSPEND_PROCESS (PROCESS:    in      NAME_STRING);
procedure SUSPEND_PROCESS (NODE:       in      NODE_TYPE);
procedure RESUME_PROCESS (PROCESS:     in      NAME_STRING);
procedure RESUME_PROCESS (NODE:        in      NODE_TYPE);
```

Suspends or resumes the designated process. SUSPEND_PROCESS can include suspension of the requesting process. While a process is suspended, the PROCESS_STATUS is SUSPENDED. RESUME causes an immediate change to the READY state. Similarly, the transition to SUSPENDED state takes place immediately.

```
function STATE_OF_PROCESS (PROCESS:    in    NAME_STRING) return PROCESS_STATUS;
function STATE_OF_PROCESS (NODE:       in    NODE_TYPE) return PROCESS_STATUS;
```

Returns the current state of the specified process. The PROCESS_STATUS of a process issuing that function will always be READY.

```
function JOB_INPUT return FILE_TYPE;

function JOB_OUTPUT return FILE_TYPE;
```

Returns the standard input or output defined at the initiation of the root process of the job. In general, these files will refer to the interactive terminal or batch input or output files, even if the current input or output file for this process has been re-directed to a different file.

6.3 PACKAGE CAIS_PROCESS_COMMUNICATION

CAIS_PROCESS_COMMUNICATION provides techniques for a process to communicate with another process or itself.

A process may send and receive inter-process messages on a number of named channels. The channels are identified by a character string with the syntax of an Ada identifier.

It is anticipated that certain channel names will eventually have standard meanings with CAIS. Each implementation must identify those channel names which have special significance.

6.3.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_PROCESS_DEFS;
with CAIS_TEXT_UTILS;
package CAIS_PROCESS_COMMUNICATION is

    subtype NODE_TYPE        is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING      is    CAIS_NODE_DEFS.NAME_STRING;
    subtype CHANNEL_STRING   is    STRING;
    subtype MESSAGE_TEXT     is    CAIS_TEXT_UTILS.TEXT;
```

```
procedure SEND (PROCESS :  in       NAME_STRING;
                CHANNEL :  in       CHANNEL_STRING;
                MESSAGE :  in       MESSAGE_TEXT;
                LIMIT :    in       DURATION := DURATION'LAST);
procedure SEND(NODE :      in out   NODE_TYPE;
                CHANNEL :  in       CHANNEL_STRING;
                MESSAGE :  in       MESSAGE_TEXT;
                LIMIT :    in       DURATION := DURATION'LAST);

procedure RECEIVE(SENDER :  in out  NODE_TYPE;
                  CHANNEL : in      STRING;
                  MESSAGE : in out  MESSAGE_TEXT;
                  LIMIT :   in      DURATION := DURATION'LAST);

 -- Exceptions

 TIME_OUT: exception;

private
    -- implementation-dependent
end CAIS_PROCESS_COMMUNICATION;
```

6.3.2 Package Semantics

```
subtype NODE_TYPE        is CAIS_NODE_DEFS.NODE_TYPE;
subtype NAME_STRING      is CAIS_NODE_DEFS.NAME_STRING;
subtype CHANNEL_STRING is STRING;
```

Provides logical name of a communication channel between communicating processes. The name is determined by mutual agreement.

```
subtype MESSAGE_TEXT is CAIS_TEXT_UTILS.TEXT;
```

The message being sent.

```
procedure SEND(PROCESS :  in       NAME_STRING;
               CHANNEL :  in       CHANNEL_STRING;
               MESSAGE :  in       MESSAGE_TEXT;
               LIMIT :    in       DURATION := DURATION'LAST);
procedure SEND(NODE :     in out   NODE_TYPE;
               CHANNEL:   in       CHANNEL_STRING;
               MESSAGE:   in       MESSAGE_TEXT;
               LIMIT:     in       DURATION := DURATION'LAST);
```

Attempts to queue up the specified MESSAGE (text) for the designated process with the specified logical CHANNEL name. If the queue is full, the calling task will be suspended up to the time LIMIT specified, after which a TIME_OUT exception is raised in the calling process. As soon as there is room for the MESSAGE, it is queued and SEND returns. It is the responsibility of the two processes to insure that whatever additional coordination required is done.

```
procedure RECEIVE(SENDER :  in out  NODE_TYPE;
                  CHANNEL : in      CHANNEL_STRING;
                  MESSAGE : in out  MESSAGE_TEXT;
                  LIMIT :   in      DURATION := DURATION'LAST);
```

Suspends the calling task until a message is available on the specified CHANNEL or the time LIMIT is reached. Multiple queued messages are received in a first-in first-out order. The capacity of the queue for a particular channel name is implementation dependent. However, before the first RECEIVE is done by a process on a particular channel name, the capacity of the queue is defined to be zero, and any SENDers will be delayed because the queue is by definition already "full." The sending process is identified by an open node handle on the process node.

6.4 PACKAGE CAIS_PROCESS_ANALYSIS

This package provides standardized debugging capabilities for processes within the CAIS implementation.

6.4.1 Package Specification

```
with CAIS_PROCESS_DEFS;
package CAIS_PROCESS_ANALYSIS is
{TBD}
end CAIS_PROCESS_ANALYSIS;
```

6.5 PACKAGE CAIS_PROCESS_INTERRUPTS

This package provides support for **pseudo-interrupts,** asynchronous signal sent between processes. Each interrupt is identified by a string with the syntax of an Ada identifier. When an interrupt is generated, the receiving process may respond by ignoring it, aborting execution, waking up a suspended task, or simply putting it on HOLD.

It is anticipated that the CAIS will define standard interrupt names, as well as standard default interrupt responses associated with each standard interrupt, in effect prior to an explicit SET_RESPONSE. The most likely default responses are ABORT for certain serious interrupts and IGNORE for all others.

Note that the predefined Ada language mechanism for associating interrupts with tasks is not being used here, so as to remain independent of any compiler implementation of this feature.

6.5.1 Package Specification

```
with CAIS_PROCESS_DEFS;
package CAIS_PROCESS_INTERRUPTS is

     subtype NODE_TYPE is CAIS_PROCESS_DEFS.NODE_TYPE;
     subtype NAME_STRING is CAIS_PROCESS_DEFS.NAME_STRING;

     subtype INTERRUPT_NAME is STRING;

     type INTERRUPT_RESPONSE is (IGNORE, ABORT, AWAKE, HOLD);

     procedure SIGNAL (PROCESS:    in      NAME_STRING;
                       INTERRUPT:  in      INTERRUPT_NAME);
     procedure SIGNAL (PROCESS:    in      NODE_TYPE;
                       INTERRUPT:  in      INTERRUPT_NAME);

     procedure SET_RESPONSE(INTERRUPT:  in      INTERRUPT_NAME;
                            RESPONSE:   in      INTERRUPT_RESPONSE;
                            TIME_LIMIT: in      DURATION := DURATION'LAST);

     function RESPONSE (INTERRUPT: in      INTERRUPT_NAME)
                   return INTERRUPT_RESPONSE;
```

```
    -- Exceptions

    USE_ERROR: exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_INTERRUPTS;
```

6.5.2 Package Semantics

```
    subtype NODE_TYPE        is CAIS_PROCESS_DEFS.NODE_TYPE;
    subtype NAME_STRING      is CAIS_PROCESS_DEFS.NAME_STRING;
    subtype INTERRUPT_NAME is STRING;
```

Typical interrupt names would be "BREAK", "HANG_UP", etc.

```
    type INTERRUPT_RESPONSE is (IGNORE, ABORT, AWAKE, HOLD);
```

This enumeration specifies the possible responses associated with an interrupt. Each interrupt has exactly one of these responses associated with it at any one time. If the response is AWAKE, then some task has executed a SET_RESPONSE (INTERRUPT_NAME, AWAKE, TIME_LIMIT) and is still suspended awaiting the interrupt signal.

```
    procedure SIGNAL (PROCESS:    in      NAME_STRING;
                      INTERRUPT:  in      INTERRUPT_NAME);
    procedure SIGNAL (PROCESS:    in      NODE_TYPE;
                      INTERRRUPT  in      INTERRUPT_NAME);
```

Generates the designated pseudo-interrupt in the named process. This call always returns immediately, even if the associated response in the receiving process is HOLD.

```
    procedure SET_RESPONSE (INTERRUPT:  in     INTERRUPT_NAME;
                            RESPONSE:   in     INTERRUPT_RESPONSE;
                            TIME_LIMIT: in     DURATION := DURATION'LAST);
```

Handles a designated pseudo-interrupt according to the designated response. If the previously set response were HOLD, and the interrupt had already occurred at least once, then the newly specified response is immediately enacted. The USE_ERROR is raised if an attempt is made to SET_RESPONSE when some other task is still suspended with the response AWAKE. In all other cases, the new response supercedes any previous default or explicitly set response.

If the response is AWAKE, then the calling task is suspended until the interrupt is received or until the time limit expires (in which case the TIME_OUT exception is raised). When setting the response to AWAKE, the previously set response is remembered, and again becomes the current response after the task is awoken due either to an interrupt or to a time-out.

```
    function RESPONSE (INTERRUPT: in       INTERRUPT_NAME)
                      return INTERRUPT_RESPONSE;
```

Indicates the current response associated with the designated interrupt for the current process. If the response is AWAKE, then some other task of the current process is suspended awaiting the interrupt.

6.6 PRAGMATICS

a. Channels — A conforming implementation must support channel names of up to 20 characters. A conforming implementation must support up to 20 simultaneous accepting channels from the same process.

7. CAIS Device Nodes

This area provides basic device input/output support, along with special device control facilities. A device must first be made accessible to an Ada program by an OPEN, specifying the external device by a NAME and a FORM, both character strings. When opening device node handles, the NAME and FORM string formats are required to be the same and refer to the same external devices in both file node usage and in the device node packages. The collection of packages in this section are defined with careful consideration of standards established for information interchange by the American National Standards Institute [ANSI77] and [ANSI79]. The interfaces are also defined with consideration for existing interactive terminals that do not conform to the ANSI standards.

7.1 VIRTUAL TERMINALS

There are three primary classes of character-imaging terminals in use today: scroll, page, and form. Four packages are provided in this section, one package for the common terminal support functions and one package for each of the three classes of terminals supported.

7.1.1 Package CAIS_TERMINAL_SUPPORT

This package provides the routines that are common to scroll, page, and form terminals.

7.1.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_TERMINAL_SUPPORT is

    type TERMINAL_TYPE is limited private;

    subtype FORM_STRING         is CAIS_NODE_DEFS.FORM_STRING;
    subtype NAME_STRING         is CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY    is CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME       is CAIS_NODE_DEFS.RELATION_NAME;

    type TERMINAL_CLASS is (SCROLL, PAGE, FORM);

    procedure CREATE (TERMINAL:   in out   TERMINAL_TYPE;
                      CLASS:      in       TERMINAL_CLASS := SCROLL;
                      NAME:       in       NAME_STRING;
                      FORM:       in       FORM_STRING := " ");
    procedure CREATE (TERMINAL:   in out   TERMINAL_TYPE;
                      CLASS:      in       TERMINAL_CLASS := SCROLL;
                      BASE:       in       NODE_TYPE;
                      KEY:        in       RELATIONSHIP_KEY := " ";
                      RELATION:   in       RELATION_NAME: = "DOT";
                      FORM:       in       FORM_STRING :=  " ");

    procedure OPEN (TERMINAL :  in out   TERMINAL_TYPE;
                    CLASS :     in       TERMINAL_CLASS := SCROLL;
                    NAME :      in       NAME_STRING;
                    FORM :      in       FORM_STRING := " ");
```

```
procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                CLASS:       in       TERMINAL_CLASS := SCROLL:
                BASE:        in       NODE_TYPE;
                KEY:         in       RELATIONSHIP_KEY := " ";
                RELATION:    in       RELATION_NAME: = "DOT";
                FORM:        in       FORM_STRING := " ");
procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                CLASS:       in       TERMINAL_CLASS := SCROLL;
                NODE:        in       NODE_TYPE;
                FORM:        in       FORM_STRING := " ");

procedure CLOSE (TERMINAL :  in out   TERMINAL_TYPE);

procedure DELETE (TERMINAL : in out   TERMINAL_TYPE);

procedure RESET (TERMINAL :  in out   TERMINAL_TYPE;
                 CLASS :     in       TERMINAL_CLASS);
procedure RESET (TERMINAL :  in out   TERMINAL_TYPE);

function CLASS(TERMINAL : in TERMINAL_TYPE) return TERMINAL_CLASS;
function NAME (TERMINAL : in TERMINAL_TYPE) return NAME_STRING;
function FORM (TERMINAL : in TERMINAL_TYPE) return FORM_STRING;


function IS_OPEN (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

type ACTIVE_POSITION is
     record
          LINE : POSITIVE;
          COLUMN : POSITIVE;
     end record;

procedure SET_POSITION (TERMINAL :  in out   TERMINAL_TYPE;
                        POSITION :  in       ACTIVE_POSITION);

function POSITION (TERMINAL : in TERMINAL_TYPE)
     return ACTIVE_POSITION;

function SIZE (TERMINAL : in TERMINAL_TYPE)
     return ACTIVE_POSITION;

-- Exceptions

CLASS_ERROR :   exception;
NAME_ERROR :    exception renames CAIS_NODE_DEFS.NAME_ERROR;
USE_ERROR :     exception renames CAIS_NODE_DEFS.USE_ERROR;
STATUS_ERROR :  exception renames CAIS_NODE_DEFS.STATUS_ERROR;

private
     -- implementation-dependent
end CAIS_TERMINAL_SUPPORT;
```

7.1.1.2 Package Semantics

```
type TERMINAL_CLASS is (SCROLL, PAGE, FORM);
```

Indicates the different classes of terminals that are supported.

```
procedure CREATE (TERMINAL:  in out   TERMINAL_TYPE;
                  CLASS:     in       TERMINAL_CLASS := SCROLL;
                  NAME:      in       NAME_STRING;
                  FORM:      in       FORM_STRING := " ");
```

Creates an extenal termnal (and its device node) that is associated with the given terminal. The given terminal is left open. A null string for the FORM specifies default options of the implementation.

The exception STATUS_ERROR is raised if the given terminal is already open. The exception NAME_ERROR is raised if the NAME does not identify an external logical terminal.

```
procedure CREATE (TERMINAL:  in out   TERMINAL_TYPE;
                  CLASS:     in       TERMINAL_CLASS := SCROLL;
                  BASE:      in       NODE_TYPE;
                  KEY:       in       RELATIONSHIP_KEY := " ";
                  RELATION:  in       RELATION_NAME: = "DOT";
                  FORM:      in       FORM_STRING := " ");
```

The semantics are the same as above except that the terminal is identified by means of BASE/KEY/RELATION.

```
procedure OPEN (TERMINAL :  in out  TERMINAL_TYPE;
                CLASS :     in      TERMINAL_CLASS := SCROLL;
                NAME :      in      NAME_STRING;
                FORM :      in      FORM_STRING := " ");
```

Associates the given terminal handle with a terminal having the given name and form and sets the current class of the terminal handle to the given class.

The exception NAME_ERROR is raised if the string given as NAME does not identify a terminal. The exception USE_ERROR is raised if the terminal identified by NAME cannot be opened in the given class or form.

```
procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                CLASS:       in       TERMINAL_CLASS := SCROLL;
                BASE:        in       NODE_TYPE;
                KEY:         in       RELATIONSHIP_KEY := " ";
                RELATION:    in       RELATION_NAME: = "DOT";
                FORM:        in       FORM_STRING := " ");
procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                CLASS:       in       TERMINAL_CLASS := SCROLL;
                NODE:        in       NODE_TYPE;
                FORM:        in       FORM_STRING := " ");
```

The semantics are the same as above except that the terminal is identified by means of the associated node or BASE/KEY/RELATION.

```
procedure CLOSE (TERMINAL :  in out   TERMINAL_TYPE);
```

Severs the association between the terminal handle and its associated terminal.

```
procedure DELETE (TERMINAL:  in out   TERMINAL_TYPE);
```

Deletes the external terminal (and its device node) associated with the given terminal. The given terminal is closed, and the external logical terminal ceases to exist.

The exception STATUS_ERROR is raised if the given terminal is not open. The exception USE_ERROR is raised if deletion of the external logical terminal is not allowed by the caller.

```
procedure RESET (TERMINAL :  in out   TERMINAL_TYPE;
                 CLASS : -   in       TERMINAL_CLASS);
procedure RESET (TERMINAL :  in out   TERMINAL_TYPE);
```

Changes the terminal handle to the given class and/or resets the terminal handle to its initial state.

```
function CLASS(TERMINAL : in TERMINAL_TYPE) return TERMINAL_CLASS;
```

Returns the class of the node associated with the given terminal handle.

```
function NAME (TERMINAL : in TERMINAL_TYPE) return NAME_STRING;
```

Returns the name of the node associated with the given terminal handle.

```
function FORM (TERMINAL : in TERMINAL_TYPE) return FORM_STRING;
```

Returns the form associated with the given terminal handle.

```
function IS_OPEN (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns TRUE if the given terminal handle is associated with a logical terminal, otherwise returns FALSE.

```
type ACTIVE_POSITION is
    record
        LINE : POSITIVE;
        COLUMN : POSITIVE;
    end record;
```

The ACTIVE_POSITION indicates the row and column position on the display of a terminal at which the next operation may occur.

```
procedure SET_POSITION (TERMINAL :  in out   TERMINAL_TYPE;
                        POSITION :  in       ACTIVE_POSITION);
```

Moves the active position to the specified POSITION on the display of the given terminal.

```
function POSITION (TERMINAL : in TERMINAL_TYPE)
    return ACTIVE_POSITION;
```

Returns the POSITION of the active position on the given terminal.

```
function SIZE (TERMINAL : in TERMINAL_TYPE)
    return ACTIVE_POSITION;
```

Returns the maximum line and maximum column of the given terminal.

7.1.2 Package CAIS_SCROLL_TERMINAL

This package provides the functionality of a common "teleprinter" type terminal. It is capable of a minimal set of operations. Characters are transmitted between a program and the terminal a character or a line at a time. This type of terminal is typically configured to echo each character as it is entered at the keyboard (before transmission to the computer or intervening communications equipment).

7.1.2.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_SCROLL_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    procedure SET_TAB (TERMINAL :      in out   TERMINAL_TYPE);

    procedure CLEAR_TAB (TERMINAL :    in out   TERMINAL_TYPE);

    procedure TAB (TERMINAL :          in out   TERMINAL_TYPE;
                   COUNT :             in       POSITIVE);

    procedure NEW_LINE (TERMINAL :     in out   TERMINAL_TYPE);

    procedure NEW_PAGE (TERMINAL :     in out   TERMINAL_TYPE);

    procedure PUT (TERMINAL :          in out   TERMINAL_TYPE;
                   ITEM :              in       CHARACTER);
    procedure PUT (TERMINAL :          in out   TERMINAL_TYPE;
                   ITEM :              in       STRING);

    procedure UPDATE (TERMINAL :       in out   TERMINAL_TYPE);

    procedure GET (TERMINAL :          in out   TERMINAL_TYPE;
                   ITEM :                 out   CHARACTER);
    procedure GET (TERMINAL :          in out   TERMINAL_TYPE;
                   ITEM :                 out   STRING);
    procedure GET (TERMINAL :          in out   TERMINAL_TYPE;
                   ITEM :                 out   STRING;
                   LAST :                 out   NATURAL);

    procedure SET_ECHO (TERMINAL :     in       TERMINAL_TYPE;
                        TO :           in       BOOLEAN := TRUE);

    function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    procedure BELL (TERMINAL :         in out   TERMINAL_TYPE);

    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_SCROLL_TERMINAL;
```

7.1.2.2 Package Semantics

```
    procedure SET_TAB (TERMINAL :   in out     TERMINAL_TYPE);
```

Creates a horizontal tab stop at the active position (used by TAB).

```
    procedure CLEAR_TAB (TERMINAL :    in out   TERMINAL_TYPE);
```

Deletes a horizontal tab stop at the active position. The exception USE_ERROR is raised if a horizontal tab stop does not exist at the active position.

```
procedure TAB (TERMINAL :     in out     TERMINAL_TYPE;
               COUNT :        in         POSITIVE);
```

Moves the active position the specified number of horizontal tab stops. The exception USE_ERROR is raised if there are fewer than COUNT tab stops on the active line.

```
procedure NEW_LINE (TERMINAL :   in out     TERMINAL_TYPE);
```

Moves the active position to the first column of the next line. The display scrolls upward if entered on the last line of the display.

```
procedure NEW_PAGE (TERMINAL :   in out     TERMINAL_TYPE);
```

Moves the active position to the first column of the first line of a new page.

```
procedure PUT (TERMINAL :     in out     TERMINAL_TYPE;
               ITEM :         in         CHARACTER);
```

Writes a single character to the display and advances the active position. If the active position is at the last column on a line, a NEW_LINE operation is performed after writing the character.

```
procedure PUT (TERMINAL :     in out     TERMINAL_TYPE;
               ITEM :         in         STRING);
```

Writes a character at a time in the same manner as PUT of a character, writing each character in the given string successively.

```
procedure UPDATE (TERMINAL :  in out     TERMINAL_TYPE);
```

Forces all data that has not already been transmitted to the terminal to be transferred.

```
procedure GET (TERMINAL :  in out     TERMINAL_TYPE;
               ITEM :         out     CHARACTER);
```

Reads a single (unedited) character from the terminal keyboard.

```
procedure GET (TERMINAL :  in out     TERMINAL_TYPE;
               ITEM :         out     STRING);
```

Reads ITEM'LENGTH (unedited) characters from the terminal keyboard into ITEM.

```
procedure GET (TERMINAL :  in out     TERMINAL_TYPE;
               ITEM :         out     STRING;
               LAST :         out     NATURAL);
```

Successively reads (unedited) characters from the terminal keyboard into ITEM, until either all positions of ITEM are filled or there are no more characters buffered for the terminal. Upon completion LAST contains the index of the last position in ITEM to contain a character that has been read.

```
procedure SET_ECHO (TERMINAL :   in         TERMINAL_TYPE;
                    TO :         in         BOOLEAN := TRUE);
```

When TO is given as TRUE, each character entered at the keyboard is echoed to the display.

```
function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether echo is enabled (TRUE) or disabled (FALSE).

```
    procedure BELL (TERMINAL :  in out       TERMINAL_TYPE);
```

Activates the bell (beeper) on the terminal.

```
    - Exceptions

    CLASS_ERROR :   exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :     exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the operations in the package CAIS_SCROLL_TERMINAL are invoked with a TERMINAL which is not OPENed or RESET with class SCROLL.

7.1.3 Package CAIS_PAGE_TERMINAL

This package provides the functionality of a page terminal. A page terminal is commonly referred to as a character-oriented or interactive terminal. This terminal may have many types of format effectors, cursor controls, and local (built-in) editing functions. Typical controls for page terminals are to position the cursor, to erase within a line or display area, to insert into or delete from a line, to insert or delete entire lines, to scroll up, and to select graphic rendition for subsequent output characters (intensity, reverse-image, blink, underscore, etc.). The terminal may be configured to echo before transmission to the computer (or intervening equipment) or not to echo at all. Each character is transmitted to the computer as it is entered at the keyboard. Except when locally echoed, the control action implied by the character keyed is deferred until (and if) the computer (or communications equipment) echoes the character. (This allows some programs, operating with non-echoing terminals, to reinterpret the meanings of control characters keyed by not directly echoing these characters. A number of popular text editors operate this way.)

7.1.3.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_PAGE_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    procedure SET_TAB (TERMINAL :                in out    TERMINAL_TYPE);

    procedure CLEAR_TAB (TERMINAL :              in out    TERMINAL_TYPE);

    procedure TAB (TERMINAL :                    in out    TERMINAL_TYPE;
                   COUNT :                       in        POSITIVE);

    procedure BELL (TERMINAL :                   in out    TERMINAL_TYPE);

    procedure DELETE_CHARACTER (TERMINAL :       in out    TERMINAL_TYPE;
                                COUNT :          in        POSITIVE);

    procedure DELETE_LINE (TERMINAL :            in out    TERMINAL_TYPE;
                           COUNT :               in        POSITIVE);

    function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    procedure ERASE_CHARACTER (TERMINAL :        in out    TERMINAL_TYPE;
                               COUNT :           in        POSITIVE);

    type SELECT_ENUM is
         (FROM_ACTIVE_POSITION_TO_END,
          FROM_START_TO_ACTIVE_POSITION,
          ALL_POSITIONS);

     procedure ERASE_IN_DISPLAY (TERMINAL :  in out   TERMINAL_TYPE;
                                 SELECTION : in       SELECT_ENUM);

     procedure ERASE_IN_LINE (TERMINAL :  in out   TERMINAL_TYPE;
                              SELECTION : in       SELECT_ENUM);

     procedure GET (TERMINAL : in out   TERMINAL_TYPE;
                    ITEM :        out   CHARACTER);
     procedure GET (TERMINAL : in out   TERMINAL_TYPE;
                    ITEM :        out   STRING;
     procedure GET (TERMINAL : in out   TERMINAL_TYPE;
                    ITEM :        out   STRING;
                    LAST :        out   NATURAL);

     procedure INSERT_CHARACTER (TERMINAL : in out   TERMINAL_TYPE;
                                 COUNT :    in       POSITIVE);

     procedure INSERT_LINE (TERMINAL : in out   TERMINAL_TYPE;
                            COUNT :    in       POSITIVE);

     procedure PUT (TERMINAL : in out   TERMINAL_TYPE;
                    ITEM :     in       CHARACTER);
     procedure PUT (TERMINAL : in out   TERMINAL_TYPE;
                    ITEM :     in       STRING);

     type GRAPHIC_RENDITION_ENUM is
          (PRIMARY_RENDITION,
           BOLD,
           FAINT,
           UNDERSCORE,
           SLOW_BLINK,
           RAPID_BLINK,
           REVERSE_IMAGE)

     procedure SELECT_GRAPHIC_RENDITION (TERMINAL : in out  TERMINAL_TYPE;
                                         SELECTION : in     GRAPHIC_RENDITION_ENUM);

     procedure SET_ECHO (TERMINAL :             in out  TERMINAL_TYPE;
                         TO :                   in      BOOLEAN := TRUE);

     procedure UPDATE (TERMINAL : in out   TERMINAL_TYPE);

     -- Exceptions

     CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
     USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
     -- implementation-dependent
end CAIS_PAGE_TERMINAL;
```

7.1.3.2 Package Semantics

```
     procedure SET_TAB (TERMINAL :   in out       TERMINAL_TYPE);
```

Creates a horizontal tab stop at the active position.

```
procedure CLEAR_TAB (TERMINAL : in out        TERMINAL_TYPE);
```

Deletes a horizontal tab stop at the active position. The exception USE_ERROR is raised if a horizontal tab stop does not exist at the active position.

```
procedure TAB (TERMINAL : in out      TERMINAL_TYPE;
               COUNT :    in          POSITIVE);
```

Moves the active position the specified number of horizontal tab stops. The exception USE_ERROR is raised if there are fewer than COUNT tab stops on the active line.

```
procedure BELL (TERMINAL : in out     TERMINAL_TYPE);
```

Activates the bell (beeper) on the terminal.

```
procedure DELETE_CHARACTER (TERMINAL : in out     TERMINAL_TYPE;
                            COUNT :    in         POSITIVE);
```

Deletes the given number of characters on the active line starting at the active position. Adjacent characters to the right of the active position are shifted left. Open space on the right is filled with SPACE characters. The active position is not changed.

```
procedure DELETE_LINE (TERMINAL : in out     TERMINAL_TYPE;
                       COUNT :    in         POSITIVE);
```

Deletes the given number of lines starting at the active line. Adjacent lines are shifted from the bottom toward the active line. COUNT lines from the bottom of the display are cleared. The active position is not changed.

```
function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether echo is enabled (TRUE) or disabled (FALSE).

```
procedure ERASE_CHARACTER (TERMINAL : in out     TERMINAL_TYPE;
                           COUNT :    in         POSITIVE);
```

Replaces the given number of characters on the active line with SPACE characters starting at the active position. The active position is not changed. The exception USE_ERROR is raised if COUNT is greater than SIZE(TERMINAL).COLUMN_POSITION(TERMINAL).COLUMN.

```
type SELECT_ENUM is
     (FROM_ACTIVE_POSITION_TO_END,
      FROM_START_TO_ACTIVE_POSITION,
      ALL_POSITIONS);
```

```
procedure ERASE_IN_DISPLAY (TERMINAL :  in out     TERMINAL_TYPE;
                            SELECTION : in         SELECT_ENUM);
```

Erases the characters in the entire display as determined by the active position and the given SELECTION (include the active position). The active position is not changed.

```
procedure ERASE_IN_LINE (TERMINAL :  in out     TERMINAL_TYPE;
                         SELECTION : in         SELECT_ENUM);
```

Erases the characters in the active line as determined by the active position and the given SELECTION (include the active position). The active position is not changed.

```
procedure GET (TERMINAL :  in out        TERMINAL_TYPE;
               ITEM :          out       CHARACTER);
```

Reads a single (unedited) character from the terminal keyboard.

```
procedure GET (TERMINAL :  in out        TERMINAL_TYPE;
               ITEM :          out       STRING);
```

Reads ITEM'LENGTH (unedited) characters from the terminal keyboard into ITEM.

```
procedure GET (TERMINAL :  in out        TERMINAL_TYPE;
               ITEM :          out       STRING;
               LAST :          out       NATURAL);
```

Successively reads (unedited) characters from the terminal keyboard into ITEM, until either all positions of ITEM are filled or there are no more characters buffered for the terminal. Upon completion LAST contains the index of the last position in ITEM to contain a character that has been read.

```
procedure INSERT_CHARACTER (TERMINAL :  in out   TERMINAL_TYPE;
                            COUNT :     in       POSITIVE);
```

Inserts COUNT SPACE characters into the active line at the active position. Adjacent characters are shifted to the right. The rightmost characters on the line may be lost. The active position is advanced to the right one character position.

```
procedure INSERT_LINE (TERMINAL :  in out   TERMINAL_TYPE;
                       COUNT :     in       POSITIVE);
```

Inserts COUNT blank lines into the display at the active line. The lines at and below the top of the display are lost. The active position remains unchanged.

```
procedure PUT (TERMINAL :  in out   TERMINAL_TYPE;
               ITEM :      in       CHARACTER);
```

Writes a single character at the active position. Advances the active position to the next column. If the character is written to the last character position on a line, advances the active position to the first column of the next line. If the character is written to the last character position of the last line, inserts a line at the bottom of the display and moves the active position to the first column of the last line.

```
procedure PUT (TERMINAL :  in out   TERMINAL_TYPE;
               ITEM :      in       STRING);
```

Writes each character of the given string according to the semantics for PUT with ITEM as a single character.

```
type GRAPHIC_RENDITION_ENUM is
      (PRIMARY_RENDITION,
       BOLD,
       FAINT,
       UNDERSCORE,
       SLOW_BLINK,
       RAPID_BLINK,
       REVERSE_IMAGE);

procedure SELECT_GRAPHIC_RENDITION (TERMINAL:  in out   TERMINAL_TYPE;
                                    SELECTION: in       GRAPHIC_RENDITION_ENUM);
```

Sets the graphic rendition for subsequent characters to be PUT. If the graphic rendition specified is not supported by the terminal, the primary rendition is used. The exception USE_ERROR is raised if the specified graphic rendition is not supported.

```
    procedure SET_ECHO(TERMINAL :   in out    TERMINAL_TYPE;
                       TO :         in        BOOLEAN := TRUE);
```

Turns on (TRUE) or off (FALSE) echoing for input file.

```
    procedure UPDATE (TERMINAL :  in out    TERMINAL_TYPE);
```

Forces all data that has not already been transmitted to the terminal to be transmitted.

```
    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the routines in the package CAIS_PAGE_TERMINAL are invoked with a terminal handle which is not OPENed or RESET with class PAGE.

7.1.4 Package CAIS_FORM_TERMINAL

This package provides functionality for manipulating a form terminal. A form terminal controls much of the display modification itself (or within local "cluster" controllers). Typically a form is built by writing control and prompting characters to desired positions on the display, setting specific character positions to be guarded (protected, as for prompts) or unguarded (unprotected, as for ill-in qualified area), and designating the attributes of the characters (legal entries, color, and intensity. The display is divided into areas of contiguous character positions (qualified area space) that have the same attributes (e.g., unprotected, high intensity). Once the form is built, the form is transmitted to the terminal. At this point, the terminal is in "local" control of the display. The user may move the cursor about on the display, insert, delete, and replace characters in any unprotected area of the display (all under local control, without use of the computer or communications circuitry). When the user has finished all the modifications/entries that are desired, the user presses a special key (function key or enter key) which causes the modified portions of the display to be accessible to the program.

7.1.4.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_FORM_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    type TERMINATION_KEY_RANGE is INTEGER
        range 0 . . implementation_defined;


    type AREA_INTENSITY is
          (NONE,
           NORMAL,
           HIGH);

    type AREA_PROTECTION is
          (UNPROTECTED,
           PROTECTED);

    type AREA_INPUT is
          (GRAPHIC_CHARACTERS,
          NUMERICS
          ALPHABETICS);
    type AREA_VALUE is
         (NO_FILL,
          FILL_WITH_ZEROES,
          FILL_WITH_SPACES);

    procedure DEFINE_QUALIFIED_AREA (TERMINAL :    in out  TERMINAL_TYPE;
                                     INTENSITY :   in      AREA_INTENSITY := NORMAL;
                                     PROTECTION :  in      AREA_PROTECTION :=
                                                           PROTECTED;
                                     INPUT :       in      AREA_INPUT :=
                                                           GRAPHIC_CHARACTER_INPUT;
                                     VALUE :       in      AREA_VALUE := NO_FILL);

    procedure CLEAR_QUALIFIED_AREA (TERMINAL :     in out  TERMINAL_TYPE);

    procedure TAB (TERMINAL :   in out  TERMINAL_TYPE;
                   COUNT:       in      POSITIVE);

    procedure PUT (TERMINAL:    in out  TERMINAL_TYPE;
                   ITEM:        in      CHARACTER);
    procedure PUT (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :       in      STRING);

    procedure ERASE_AREA (TERMINAL :       in out   TERMINAL_TYPE);

    procedure ERASE_DISPLAY (TERMINAL :    in out   TERMINAL_TYPE);

    procedure ACTIVATE_FORM (TERMINAL :    in out   TERMINAL_TYPE);

    procedure GET (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :          out  CHARACTER);
    procedure GET (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :          out  STRING);

    function IS_FORM_UPDATED (TERMINAL :    in     TERMINAL_TYPE return BOOLEAN;

    function TERMINATION_KEY (TERMINAL : in TERMINAL_TYPE) return TERMINATION_KEY_RANGE;

    function AREA_QUALIFIER_REQUIRES_SPACE (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_FORM_TERMINAL;
```

7.1.4.2 Package Semantics

```
    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    type TERMINATION_KEY_RANGE is INTEGER range 0 . . implementation_defined;
```

```
type AREA_INTENSITY is
        (NONE,
         NORMAL,
         HIGH);

type AREA_PROTECTION is
        (UNPROTECTED,
         PROTECTED);

type AREA_INPUT is
        (GRAPHIC_CHARACTERS,
         NUMERICS,
         ALPHABETICS);

type AREA_VALUE is
        (NO_FILL,
         FILL_WITH_ZEROES,
         FILL_WITH_SPACES);
```

These types define the attributes for a qualified area of a form. AREA_INTENSITY indicates the intensity at which the characters in the area should be displayed (NONE indicates that characters are not displayed). AREA_PROTECTION specifies whether the user can modify the contents of the area when the form has been activated. AREA_INPUT specifies the valid characters that may be entered by the user (GRAPHIC_CHARACTERS indicates that any printable character may be entered). AREA_VALUE indicates the initial value that the area should have when activated (NO_FILL indicates that the value has been specified by a previous PUT statement).

```
procedure DEFINE_QUALIFIED_AREA (TERMINAL:    in out   TERMINAL_TYPE;
                                 INTENSITY:   in       AREA_INTENSITY: = NORMAL;
                                 PROTECTION:  in       AREA_PROTECTION: = PROTECTED;
                                 INPUT:       in       AREA_INPUT : =
                                                       GRAPHIC_CHARACTER_INPUT;
                                 VALUE:       in       AREA_VALUE : = NO_FILL);
```

Indicates that the active position is the first character position of a qualified area. The end of the qualified area is indicated by the beginning of the following qualified area.

```
procedure CLEAR_QUALIFIED_AREA (TERMINAL :    in out   TERMINAL_TYPE);
```

Removes an area qualifier from the active position.

```
procedure TAB (TERMINAL :   in out     TERMINAL_TYPE;
               COURT :      in         POSITIVE);
```

Moves the active position the specified number of qualified areas toward the end of the display. The active position is the first character position of the designated qualified area. The exception USE_ERROR is raised if there are fewer than COUNT qualified areas after the active position.

```
procedure PUT (TERMINAL :   in out     TERMINAL_TYPE;
               ITEM :       in         CHARACTER);
```

Writes a character to the display in the active position. The column of the active position is incremented by one. If the character is written in the last column of a line, the active position is advanced to the first column of the following line. If the character is written to the last column of the last line, the active position is moved to the first column of the first line. If the area qualifier takes space on the display, writing to the position containing an area qualifier removes the area qualifier. Only characters in the range SPACE through STANDARD.ASCII.TILDE may be written. An attempt to write any other character raises the USE_ERROR exception.

```
procedure PUT (TERMINAL :   in out     TERMINAL_TYPE;
               ITEM :       in         STRING);
```

Writes each character of the ITEM according to the semantics for writing an individual character.

```
procedure ERASE_AREA (TERMINAL :   in out     TERMINAL_TYPE);
```

Clears the area in which the active position is located.

```
procedure ERASE_DISPLAY (TERMINAL : in out     TERMINAL_TYPE);
```

Clears the display and removes all area qualifiers.

```
procedure ACTIVATE_FORM (TERMINAL :in out     TERMINAL_TYPE);
```

Activates the form that has been created enabling the user to edit the form. Returns control to the calling task when user enters a termination key.

```
procedure GET (TERMINAL :   in out     TERMINAL_TYPE;
               ITEM :          out     CHARACTER);
```

Reads a character from the display at the active position. Advances the active position forward one position. An area qualifier (on a display on which the area qualifier requires space) is read as the SPACE character.

```
procedure GET (TERMINAL :   in out     TERMINAL_TYPE;
               ITEM :          out     STRING);
```

Reads ITEM'LENGTH characters from the display one at a time filling the ITEM from ITEM'FIRST through ITEM'LAST.

```
function IS_FORM_UPDATED (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether the form was modified by the user during the previous ACTIVATE_FORM operation.

```
function TERMINATION_KEY (TERMINAL : in TERMINAL_TYPE) return TERMINATION_KEY_RANGE;
```

Returns a number that indicates which (implementation dependent) key terminated the ACTIVATE_FORM procedure. A value of zero indicates the normal termination key (i.e., the ENTER key).

```
function AREA_QUALIFIER_REQUIRES_SPACE (TERMINAL : in TERMINAL_TYPE)
     return BOOLEAN;
```

Returns TRUE if the area qualifier requires space on the display.

```
-- Exceptions

CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the routines in the package CAIS_FORM_TERMINAL are invoked with a terminal handle which is not OPENed or RESET with class FORM.

7.2 PACKAGE CAIS_DEVICE_CONTROL

This package provides physical device control interfaces. For each device type, there is a set of operations defined to manipulate the device.

Certain generic device-oriented status information is available outside of the specific packages.

7.2.1 Package Specification

```
package CAIS_DEVICE_CONTROL is
        {TBD}
 end CAIS_DEVICE_CONTROL;
```

8. CAIS UTILITIES

This area provides packages for manipulating strings and parameter lists. It also defines additional pragmatic requirements for a conforming implementation of the predefined Ada LRM packages.

8.1 PREDEFINED LANGUAGE ENVIRONMENT

The facilities described in the Ada LRM that are used directly by the CAIS include the packages STANDARD and SYSTEM, as discussed in the following subsections. See the Pragmatics Section 8.3.

8.1.1 Package STANDARD

Package STANDARD forms the outermost scope of all Ada compilation units.

Package STANDARD is not replaceable by implementors of the CAIS, and hence the "CAIS_" prefix is not used.

8.1.2 Package SYSTEM

The package SYSTEM is provided as a language-defined package which defines certain parameters of the language implementation.

Package SYSTEM is not replaceable by implementors of the CAIS, and hence the "CAIS_" prefix is not used.

8.2 PREDEFINED UTILITY PACKAGES

The utilities necessary for the support of other CAIS interfaces include the packages CAIS_TEXT_UTILS and CAIS_LIST_UTILS, as discussed in the following sections.

8.2.1 Package CAIS_TEXT_UTILS

This package implements basic operations on a string type which is of dynamic length. It defines the type used to implement lists and is used for MESSAGE_TEXT, PROCESS_STRING, and RESULTS_STRING.

8.2.1.1 Package Specification

```
package CAIS_TEXT_UTILS is
     MAXIMUM : constant : = implementation_defined;
     subtype INDEX is INTEGER range 0...MAXIMUM;

     type TEXT is limited private;

     function LENGTH  (T: TEXT) return INDEX;
     function VALUE   (T: TEXT) return STRING;
     function EMPTY   (T: TEXT) return BOOLEAN;
```

```
    procedure INIT_TEXT(T:  in out TEXT);
    procedure FREE_TEXT(T:  in out TEXT);

    function TO_TEXT (S: STRING)        return TEXT;
    function TO_TEXT (C: CHARACTER)     return TEXT;

    function "&"  (LEFT: TEXT;        RIGHT: TEXT)        return TEXT;
    function "&"  (LEFT: TEXT;        RIGHT: STRING)      return TEXT;
    function "&"  (LEFT: STRING;      RIGHT: TEXT)        return TEXT;
    function "&"  (LEFT: TEXT;        RIGHT: CHARACTER) return TEXT;
    function "&"  (LEFT: CHARACTER; RIGHT: TEXT)          return TEXT;

    function "="    (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function "<="   (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function "<"    (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function ">="   (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function ">"    (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;

    procedure SET   (OBJECT: in out TEXT;  VALUE: in TEXT);
    procedure SET   (OBJECT: in out TEXT;  VALUE: in STRING);
    procedure SET   (OBJECT: in out TEXT;  VALUE: in CHARACTER);

    procedure APPEND  (TAIL: in TEXT;         TO: in out TEXT);
    procedure APPEND  (TAIL: in STRING;       TO: in out TEXT);
    procedure APPEND  (TAIL: in CHARACTER;    TO: in out TEXT);

    procedure AMEND   (OBJECT:    in out TEXT;
                       BY:        in     TEXT;
                       POSITION:  in     INDEX);
    procedure AMEND   (OBJECT:    in out TEXT;
                       BY:        in     STRING;
                       POSITION:  in     INDEX);
    procedure AMEND   (OBJECT:    in out TEXT;
                       BY:        in     CHARACTER;
                       POSITION:  in     INDEX);

    function LOCATE  (FRAGMENT: TEXT;         WITHIN: TEXT) return INDEX;
    function LOCATE  (FRAGMENT: STRING;       WITHIN: TEXT) return INDEX;
    function LOCATE  (FRAGMENT: CHARACTER; WITHIN: TEXT) return INDEX;

private
    -- implementation-dependent
end CAIS_TEXT_UTILS;
```

8.2.1.2 Package Semantics

```
    type TEXT is limited private;
```

The type is made limited private because it may be reference counted and automatically freed at last use.

```
    function LENGTH   (T: TEXT) return INDEX;
    function VALUE    (T: TEXT) return STRING;
    function EMPTY    (T: TEXT) return BOOLEAN;
```

Provides text string functions.

```
    procedure INIT_TEXT(T: in out TEXT);
```

Creates a null string.

```
procedure FREE_TEXT(T: in out TEXT);
```

Frees a string.

```
function TO_TEXT (S: STRING)          return TEXT;
function TO_TEXT (C: CHARACTER)       return TEXT;
```

Converts the given string or characters to text.

```
function "&"  (LEFT: TEXT;        RIGHT: TEXT)        return TEXT;
function "&"  (LEFT: TEXT;        RIGHT: STRING)      return TEXT;
function "&"  (LEFT: STRING;      RIGHT: TEXT)        return TEXT;
function "&"  (LEFT: TEXT;        RIGHT: CHARACTER)   return TEXT;
function "&"  (LEFT: CHARACTER;   RIGHT: TEXT)        return TEXT;
```

Concatenates to text.

```
function "="   (LEFT: TEXT;   RIGHT: TEXT)   return BOOLEAN;
function "<="  (LEFT: TEXT;   RIGHT: TEXT)   return BOOLEAN;
function "<"   (LEFT: TEXT;   RIGHT: TEXT)   return BOOLEAN;
function ">="  (LEFT: TEXT;   RIGHT: TEXT)   return BOOLEAN;
function ">"   (LEFT: TEXT;   RIGHT: TEXT)   return BOOLEAN;
```

Provides indicated comparison functions.

```
procedure SET   (OBJECT: in out TEXT; VALUE: in TEXT);
procedure SET   (OBJECT: in out TEXT; VALUE: in STRING);
procedure SET   (OBJECT: in out TEXT; VALUE: in CHARACTER);
```

Sets the object to the given value.

```
procedure APPEND   (TAIL: in TEXT;        TO: in out TEXT);
procedure APPEND   (TAIL: in STRING;      TO: in out TEXT);
procedure APPEND   (TAIL: in CHARACTER;   TO: in out TEXT);
```

Appends the given TAIL to the TO TEXT.

```
procedure AMEND   (OBJECT:     in out TEXT;
                   BY:         in     TEXT;
                   POSITION:   in     INDEX);
procedure AMEND   (OBJECT:     in out TEXT;
                   BY:         in     STRING;
                   POSITION:   in     INDEX);
procedure AMEND   (OBJECT:     in out TEXT;
                   BY:         in     CHARACTER;
                   POSITION:   in     INDEX);
```

Replaces part of the OBJECT by the given TEXT, STRING, or CHARACTER starting at the given position in the OBJECT.

```
function LOCATE (FRAGMENT:   TEXT; WITHIN: TEXT) return INDEX;
function LOCATE (FRAGMENT:   STRING; WITHIN: TEXT) return INDEX;
function LOCATE (FRAGMENT:   CHARACTER; WITHIN: TEXT) return INDEX;
```

Returns the INDEX of the FRAGMENT within the given TEXT.

8.2.2 Package CAIS_LIST_UTILS

This package is generally useful for the manipulation of all lists built following the CAIS parameter list conventions: a parenthesized, comma-separated list of items, each item in the form of a list, a string without embedded spaces or separators, or a quoted string following the Ada syntax rules, optionally preceded by a keyword identifier and a "right arrow." This syntax roughly corresponds to the Ada syntax for aggregates or for subprogram calling sequences. An approximate BNF for the CAIS list is as follows:

```
LIST             ::=  '(' [ KEYWORD '=  ' ] ITEM { ' , ' [ KEYWORD '=  ' ] ITEM } ')'
ITEM             ::=  IDENTIFIER | NUMBER | LIST | QUOTED_STRING
KEYWORD          ::=  IDENTIFIER
QUOTED_STRING    ::=  ' " ' { NON_QUOTE_CHARACTER | ' " " ' } ' " '
```

The package CAIS_LIST_UTILS uses the TEXT type defined within CAIS_TEXT_UTILS and defines additional operations. It defines the type list which is used to represent CAIS_ATTRIBUTE values.

8.2.2.1 Package Specification

```
with CAIS_TEXT_UTILS;
package CAIS_LIST_UTILS is

    type COUNT is range 0 . . implementation_defined;
    subtype POSITIVE_COUNT is COUNT range 1 . . COUNT'LAST;
    subtype LIST is CAIS_TEXT_UTILS.TEXT;
    subtype KEY_STRING is STRING;
    type ITEM_KIND is (LIST, IDENTIFIER, NUMBER, QUOTED_STRING);

    procedure INIT_LIST(L: in out LIST);

    procedure FREE_LIST(L: in out LIST);
    function  IS_EMPTY  (L: in LIST)  return BOOLEAN;
    function KIND   (L: in LIST)   return ITEM_KIND;

    function QUOTED_STRING  (L: in LIST)   return STRING;
    function IDENTIFIER     (L: in LIST)   return STRING;
    function NUMBER         (L: in LIST)   return INTEGER;

    procedure TO_LIST_QUOTED       (L: in out LIST; FROM: STRING);

    procedure TO_LIST              (L: in out LIST; FROM: STRING);
    procedure TO_LIST              (L: in out LIST; FROM: INTEGER);

    procedure SET (L: in out LIST; VALUE in LIST);
    function NUM_POSITIONAL  (L: LIST) return COUNT;

    procedure ADD_POSITIONAL  (L:     in out LIST;
                               ITEM:  in     LIST);
    procedure ADD_POSITIONAL  (L:     in out LIST;
                               ITEM:  in     STRING);

    procedure GET_POSITIONAL  (L:     in     LIST;
                               ITEM:  in out LIST;
                               AT:    in     POSITIVE_COUNT);

    procedure SET_POSITIONAL (L:     in out LIST;
                              ITEM:  in     LIST;
                              AT:    in     POSITIVE_COUNT);
    function NUM_NAMED (L: LIST)     return COUNT;
    procedure ADD_NAMED   (L:         in out LIST;
                            KEYWORD:  in     KEY_STRING;
                            ITEM:     in     LIST);
    procedure ADD_NAMED   (L:         in out LIST;
                            KEYWORD:  in     KEY_STRING;
                            ITEM:     in     STRING);

    procedure GET_NAMED   (L:         in     LIST;
                            ITEM:     in out LIST;
                            AT:       in     KEY_STRING);
    procedure GET_NAMED   (L:         in     LIST;
                            ITEM:     in out LIST;
                            AT:       in     POSITIVE_COUNT);

    procedure  SET_NAMED  (L:         in out    LIST;
                            ITEM:     in        LIST;
                            AT:       in        KEY_STRING);
    procedure  SET_NAMED  (L:         in out    LIST;
                            ITEM:     in        LIST;
                            AT:       out       POSITIVE_COUNT);

    function KEYWORD  (L:          in    LIST;
                       AT:         in    POSITIVE_COUNT)
                       return KEY_STRING;

private
    -- implementation-dependent
end CAIS_LIST_UTILS;
```

8.2.2.2 Package Semantics

```
    type ITEM_KIND is (LIST, IDENTIFIER, NUMBER, QUOTED_STRING);
```

Each item is recognizable as a list, identifier, number, or quoted-string.

```
    procedure INIT_LIST   (L: in out LIST);
```

Creates a null LIST.

```
    procedure FREE_LIST   (L: in out LIST);
```

Frees a LIST.

```
    function IS_EMPTY  (L: in      LIST) return BOOLEAN;
```

Returns TRUE if the list is an empty LIST.

```
    function KIND   (L: in      LIST) return ITEM_KIND;
```

Returns ITEM_KIND of LIST.ITEM_KIND is LIST for empty LIST.

```
    function QUOTED_STRING   (L: in LIST)   return STRING;
    function IDENTIFIER      (L: in LIST)   return STRING;
    function NUMBER          (L: in LIST)   return INTEGER;
```

Converts from a LIST according to the ITEM_KIND.

```
procedure TO_LIST_QUOTED  (L: in out LIST; FROM: STRING);

procedure TO_LIST         (L: in out LIST; FROM: STRING);
procedure TO_LIST         (L: in out LIST; FROM: INTEGER);
```

Converts to a LIST according to the ITEM_KIND.

```
procedure SET  (L: in out LIST; VALUE: in LIST);
```

Sets the LIST L to the given VALUE.

```
function NUM_POSITIONAL(L: LIST) return COUNT;
```

Returns COUNT of positional components (i.e., those without the "KEYWORD => " part).

```
procedure ADD_POSITIONAL  (L:     in out LIST;
                           ITEM:  in     LIST);
procedure ADD_POSITIONAL  (L:     in out LIST;
                           ITEM:  in     STRING);
```

Adds another ITEM to the end of the LIST of positional components.

```
procedure GET_POSITIONAL  (L:     in     LIST;
                           ITEM:  in out LIST;
                           AT:    in     POSITIVE_COUNT);
```

Retrieves ITEM at specified position of LIST. Returns empty LIST if AT > NUM_POSITONAL(L).

```
procedure SET_POSITIONAL  (L:     in out LIST;
                           ITEM in       LIST
                           AT:    in     POSITIVE_COUNT);
```

Sets VALUE at specified position of LIST to the given ITEM.

```
function NUM_NAMED  (L: LIST) return COUNT;
```

Returns count of named components (i.e., those with the "KEYWORD =>" part).

```
procedure ADD_NAMED (L:          in out LIST;
                     KEYWORD:    in     KEY_STRING;
                     ITEM:       in     LIST);
procedure ADD_NAMED (L:          in out LIST;
                     KEYWORD:    in     KEY_STRING;
                     ITEM:       in     STRING);
```

Adds another named ITEM to LIST. An exception is generated if an ITEM with the given KEYWORD already exists within LIST.

```
procedure GET_NAMED (L:       in     LIST;
                     ITEM:    in out LIST;
                     AT:      in     KEY_STRING);
procedure GET_NAMED (L:       in     LIST;
                     ITEM:    in out LIST;
                     AT:      in     POSITIVE_COUNT);
```

Gets the named ITEM at the given KEYWORD or POSITIVE_COUNT; returns empty LIST if the ITEM is not found.

```
procedure SET__NAMED (L:        in out LIST;
                      ITEM:     in     LIST;
                      AT:       in     KEY__STRING);
procedure SET__NAMED (L:        in out LIST;
                      ITEM:     in     LIST;
                      AT:       in     POSITIVE__COUNT
```

Sets the named component at the given KEY__STRING or POSITIVE__COUNT to the given ITEM.

```
function KEYWORD  (L:  in    LIST;
                   AT: in    POSITIVE__COUNT)
                   return KEY__STRING;
```

Returns the KEYWORD of the specified named item.

8.2.3 Package CAIS__HELP__UTILS

This package provides standard support for help facilities.

{TBD}

8.3 PRAGMATICS

a.	STANDARD	The CAIS places certain requirements on the pre-defined types available. In particular, a conforming implementation must support some integer type with at least the range -32767 to 32767.
b.	SYSTEM	The CAIS places certain requirements on the machine parameters. In particular, a conforming implementation must have MIN__INT = -32767 and MAX__INT = 32767.
c.	CAIS__TEXT__UTILS	A conforming implementation must support strings of at least 32767 characters in length.

APPENDIX A
NOTES AND EXPLANATIONS

A.1 INTRODUCTION

This appendix is provided to give the reader a perspective of the context in which the CAIS is expected to function and some of the design considerations included during the CAIS generation process. While Version 1.1 of the CAIS is directed toward the DoD AIE and ALS developments, the goal of future versions of the CAIS is to provide a standard for DoD APSEs.

A.1.1 BACKGROUND

Version 1.1 of the CAIS is predicated on four premises:

1) the CAIS will be implementable on the AIE
2) the CAIS will be implementable on the ALS
3) the CAIS will be implementable on a bare machine
4) the CAIS will be compatible with modern operating systems

The CAIS as described in Version 1.1 has strived to retain these perspectives while establishing a sufficiently flexible structure that can be evolved into a Version 2.0 document. This structure is believed to be flexible enough to provide CAIS implementors considerable amplitude in selecting specific approaches for actual implementations. Interference with implementation strategies has been avoided.

A.2 CONTEXT

The CAIS applies to Ada Programming Support Environments [STONEMAN] which are to become the basic software development environments for DoD development programs. Those Ada programs that are used in support of software development are defined as **tools**. This includes the spectrum of support software from project management through code development, configuration management and life-cycle support. Tools are not restricted to only those software items normally associated with program generation such as editors, compilers, debuggers, and linker-loaders. Those tools that are composed of a number of independent but inter-related programs (such as a debugger which is related to a specific compiler) are classed as **toolsets**. In this document the terms tool and toolset are used interchangeability.

Since the goal of the CAIS is to promote interoperability and transportability of Ada software across DoD APSEs, the following definitions of these terms are provided. **Interoperability** is defined as "the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion." **Transportability** of an APSE tool is defined as "the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonymously." [Reference: KAPSE Interface Team: Public Report, Volume I, 1 April 1982; p. C1].

APPENDIX B
PROVIDING DIRECTORY STRUCTURES USING A TRANSITIONAL SUBSET OF THE CAIS

B.1 INTRODUCTION

While conformance with the CAIS will be measured on a package-by-package basis, it is sometimes not possible to implement one package without the provision of another. This is especially true for packages depending on the package CAIS_NODE_MANAGEMENT. In the interest of the availability of CAIS implementations within a very short time frame, a transitional subset of the node-related packages are defined in this appendix. They include the most important interfaces that are vital for the majority of simple tools. This subset restricts the model of the file organization to the equivalent of a hierarchical tree-oriented file-system. Leaves in the tree are file nodes; all other nodes are structural nodes representing directories or they are process nodes.

In order to prevent incompatibilities with more sophisticated CAIS implementations, the syntactic appearance and semantic meaning of calls on CAIS interfaces have been kept upward compatible, rather than providing more appropriate mnemonic names for the subprograms. (The latter is left to a trivial renaming package outside the CAIS subset.) Hence, any program executing properly on an implementation of the CAIS subset will also execute properly on any implementation of the CAIS (but obviously not vice-versa).

An implementator of these transitional subset packages may choose to use different implementation strategies than required for the provision of the full functionality of these packages as defined in the CAIS.

The subset is obtained by imposing restrictions and adjusting package specifications as follows:

1. Pathnames are allowed to contain only path elements referring to the "DOT"-relation using the abbreviated form " . " or to " 'USER" and " 'CURRENT_USER" as predefined optional prefixes to pathnames.

2. In all subprograms of the node-related packages CAIS_NODE_MANAGEMENT and CAIS_STRUCTURAL_NODES any occurrence of a formal parameter of type RELATION_NAME is deleted. The implementation of these subprograms must default the RELATION_NAME to "DOT".

3. The formal parameters RELATION and PRIMARY_ONLY of the subprograms CAIS_NODE_MANAGEMENT.ITERATE are deleted. The implementation of the subprograms must default the RELATION to "DOT".

4. The following subprograms of the package CAIS_NODE_MANAGEMENT are defined to raise the USE_ERROR exception:

   PRIMARY_RELATION
   PATH_KEY
   PATH_RELATION
   LINK
   UNLINK

5. The following subprograms of the package CAIS_STRUCTURAL_NODE are defined to raise the USE_ERROR exception:

   CREATE_NODE with formal parameter "RELATION" (two instances)

Bearing these restrictions in mind, the specified semantics for all subprograms of the packages involved describe those operations useful in particular for handling directories (structural nodes) of a conventional tree-structured file system and files contained in such directories. Pathnames have the conventional form of identifiers separated by dots, except for the optional prefix path elements " 'USER " and " 'CURRENT_USER".

B.1.1 Package Semantics

NOTE: These semantics do not include the procedures and functions which are defined to raise USE_ERROR in the above list.

a) CAIS_STRUCTURAL_NODES

```
procedure CREATE_NODE(NAME:   in       NAME_STRING;
                      FORM:   in       FORM_STRING := " ");
procedure CREATE_NODE(NODE:   in out   NODE_TYPE;
                      NAME:   in       NAME_STRING;
                      FORM:   in       FORM_STRING := " ");
```

Creates a directory (structural node) with its "DOT" relationship and parent node implied by the NAME argument.

b) CAIS_NODE_MANAGEMENT

The key of a file or directory is the relationship key of the last element of its pathname. Many operations are allowed to take either a pathname, or a parent node (i.e., a directory) and a key. The keys of process nodes, file nodes or sub-directories in a directory must be mutually distinct.

```
procedure OPEN (NODE:   in out   NODE_TYPE;
                NAME:   in       NAME_STRING;
procedure OPEN (NODE:   in out   NODE_TYPE;
                BASE:   in       NODE_TYPE;
                KEY:    in       RELATIONSHIP_KEY := " ");
```

Opens the designated file node, process node or directory and returns an open handle on the designated file node, process node or directory node. The NAME_ERROR exception will be raised if the file, process or directory *does not exist.*

```
procedure CLOSE(NODE: in out NODE_TYPE);
```

Severs any association between the internal node handle and an external node and releases any associated lock. This must be done before another OPEN can be done using the same NODE_TYPE variable.

```
function IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;
```

Returns TRUE if the NODE is open.

```
function KIND (NODE: in NODE_TYPE) return CAIS_NODE_DEFS.FILE_KIND;
```

Returns the "kind" of a node, either FILE, PROCESS, STRUCTURAL or DEVICE. Structural nodes are directories.

```
function PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;
```

Returns the full path name to the file node, process node, or directory.

```
function PRIMARY_KEY (NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
```

Return the last relationship key of the pathname to the file node, process node or directory. If the NODE is a top-level directory, the key is the user name.

```
procedure GET_PARENT(NODE:     in       NODE_TYPE;
                     PARENT:   in out   NODE_TYPE);
```

Returns the parent process or directory. Generates an exception if NODE is a top-level directory.

```
procedure COPY_NODE (FROM:       in       NODE_TYPE;
                     TO:         in       NAME_STRING);
procedure COPY_NODE (FROM:       in       NODE_TYPE;
                     TO_BASE:    in       NODE_TYPE;
                     TO_KEY:     in       RELATIONSHIP_KEY := " ");
```

Copies a file. It is an error (KIND_ERROR) if the node referenced is a process node or a device node or directory node(structural node).

```
procedure COPY_TREE (FROM:       in       NODE_TYPE;
                     TO:         in       NAME_STRING);
procedure COPY_TREE (FROM:       in       NODE_TYPE;
                     TO_BASE:    in       NODE_TYPE;
                     TO_KEY:     in       RELATIONSHIP_KEY := " ");
```

Copies a directory including its files. It is an error (KIND_ERROR) if any node referenced is a process node or a device node.

```
procedure RENAME(NODE:                 in       NODE_TYPE;
                 NEW_NAME:             in       NAME_STRING);
procedure RENAME(NODE:                 in       NODE_TYPE;
                 NEW_BASE:             in       NODE_TYPE;
                 NEW_KEY:              in       RELATIONSHIP_KEY := " "
                 NEW_RELATION:         in       RELATION_NAME : = " . ";
```

Allows the renaming of file nodes, process nodes, or directories using a node handle for the renamed node and, in the second case, a node handle on the parent directory or process node. RENAME raises the exception USE_ERROR if a node already exists with the new_name.

```
procedure DELETE_NODE (NODE: in out NODE_TYPE);
procedure DELETE_NODE (NAME: in       NAME_STRING);
```

Deletes the relationships between a file or process node and its parent and deletes the node itself. This is only legal if the node has no children. Deletes a file, empty directory or a process with no descendants as well as the associated node.

```
procedure DELETE_TREE (NODE: in out NODE_TYPE);
```

DELETE_TREE deletes a node and recursively deletes all its descendants.

```
type NODE_ITERATOR is private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
```

RELATIONSHIP_KEY_PATTERNs follow the syntax of relationship keys, except that a "?" will match any single character and a "*" will match any string of characters.

```
procedure ITERATE(ITERATOR:            out      NODE_ITERATOR;
                  NODE:            in           NODE_TYPE;
                  KIND:            in           NODE_KIND;
                  KEY:             in           RELATIONSHIP_KEY_PATTERN := " * ");

function MORE (ITERATOR: in NODE_ITERATOR) return BOOLEAN;

procedure GET_NEXT(ITERATOR:       in out   NODE_ITERATOR;
                   NEXT_NODE:      in out   NODE_TYPE);
```

These three routines iterate through those nodes referred to from the given NODE, via "DOT"-relationships, that have keys satisfying the specified patterns and are of the KIND specified.

The nodes are returned in ASCII lexicographical order by relationship KEY. The key is available from the function PRIMARY__KEY (see above).

```
    procedure SET__CURRENT__NODE(NAME: in NAME__STRING);
    procedure SET__CURRENT__NODE(NODE: in NODE__TYPE);
```

Specifies NODE/NAME as the current directory.

```
    procedure GET__CURRENT__NODE(NODE: in out NODE__TYPE);
```

Associates NODE with the current directory.

```
    function IS__SAME(NAME1:          in      NAME__STRING;
                      NAME2:          in      NAME__STRING)
       return BOOLEAN;
    function IS__SAME(NODE1:          in      NODE__TYPE
                      NODE2:          in      NOTE__TYPE)
       return BOOLEAN;
```

APPENDIX C
CAIS IMPLEMENTABILITY

C.1 INTRODUCTION

The specification of the CAIS has been separated into multiple packages to simplify initial or partial implementations. The rules for Ada limited private types can interfere with this kind of separation. This appendix outlines several implementation approaches which are consistent with both the rules of the Ada language and the rules for CAIS conformance. This appendix will ultimately be superceded by a CAIS implemator's guide.

(a) NESTED GENERIC SUBPACKAGES IMPLEMENTATION

This implementation strategy seeks to minimize visibility of the limited private types of CAIS_NODE_DEFS by using these private types strictly as intended by Ada. All operations on the private types are encapsulated within the package defining CAIS_NODE_DEFS. A sketch of this is as follows:

```
package CAIS is
    -- type definitions of CAIS_NODE_DEFS
        generic
        package NODE_DEFS is
        -- subtype Declarations
        end NODE_DEFS;

        generic
        package NODE_MANAGEMENT is
        --specifications of Section 3.5
        end NODE_MANAGEMENT;

        generic
        package STRUCTURAL_NODES is
        --specifications of Section 4.1
        end STRUCTURAL_NODES;
        -- and so forth for all of the CAIS packages

end CAIS;

with CAIS;

package CAIS_NODE_DEFS is new CAIS.NODE_DEFS;

with CAIS;
package CAIS_NODE_MANAGEMENT is new CAIS.NODE_MANAGEMENT;

. . . for each of the CAIS packages
```

This organization, while unwieldy, allows the CAIS packages specified in this document to be utilized in the organization provided in earlier document sections.

(b) LIMITED RECORD TYPE IMPLEMENTATION

This sketch shows how an implementor might separate the limited private definitions and operations on the limited private types into a separate isolated package. The user-visible package structure remains the same, except that NODE_TYPE is defined as a limited record type, rather than limited private.

```
package CAIS_PRIVATE is
    type NODE_TYPE is limited private;
    . . . and other types with limited private visibility needs

    -- The remainder of this package specification is
    -- implementation specific, and not specified as part
    -- of the CAIS. No tool or APSE application should
    -- make use of this package; it is solely for the
    -- use for implementation of other CAIS packages.

end CAIS_PRIVATE;

with CAIS_PRIVATE;
package CAIS_NODE_DEFS is
    type NODE_TYPE is
      record
        INTERNALS: CAIS_PRIVATE.NODE_TYPE;
      end record;
    . . . and the rest of CAIS_NODE_DEFS from 3.1
```

The implementation of the other CAIS packages (i.e., the package bodies) may now use the underlying subprograms of CAIS_PRIVATE to manipulate the INTERNALS of NODE_TYPE. This provides an implementation which is safe, so long as no tool or applications program "withs" CAIS_PRIVATE.

A typical CAIS implementation package body may have the following appearance:

```
with CAIS_PRIVATE;
package body CAIS_NODE_MANAGEMENT is
. . .
procedure OPEN(NODE:                in out  NODE_TYPE;
               NAME:                in      NAME_STRING) is
    begin
      CAIS_PRIVATE.OPEN(NODE.INTERNALS, NAME);
    end OPEN;

. . .

end CAIS_NODE_MANAGEMENT;
```

(c) NON-ADA IMPLEMENTATION

If the package bodies are implemented in a language other than Ada, then the problems of limited private types may be absent. The implementation may have a structure dictated by the facilities of an underlying operating system, by the facilities of a microcoded system and by the processor architecture itself.

Postscript : Submission of Comments

For submission of comments on this CAIS Version 1.1, we would appreciate them being sent by Arpanet to the address

CAIS-COMMENT at ECLB

If you do not have Arpanet access, please send the comments by mail

Mr. Jack Foidl
TRW SYSTEMS
3420 Kenyon St.
Suite 202
San Diego, CA 92110

For mail comments, it will assist us if you are able to send them on 8-inch single-sided single-density DEC format diskette— but even if you can manage this, please also send us a paper copy, in case of problems with reading the diskette.

All comments are sorted and processed mechanically in order to simplify their analysis and to facilitate giving them proper consideration. To aid this process you are kindly requested to precede each comment with a three line header

```
!section . . .
!version 1983
!topic . . .
!rationale
```

The section line includes the section number, your name or affiliation (or both), and the date in ISO standard form (year-month-day). As an example, here is the section line of comment 1194 on a previous version:

```
!section 03.02.01(12)D.Taffs 82-04-26
```

The version line, for comments on the current document, should only contain "!version 1983". Its purpose is to distinguish comments that refer to different versions.

The topic line should contain a one line summary of the comment. This line is essential, and you are kindly asked to avoid topics such as "Typo" or "Editorial comment" which will not convey any information when printed in a table of contents. As an example of an informative topic line, consider:

```
!topic FILE NODE MANAGEMENT
```

Note also that nothing prevents the topic line from including all the information of a comment, as in the following topic line:

```
!topic Insert: "...are {implicitly} defined by a subtype declaration"
```

The rationale line should contain some reasoning for your comment.

As a final example here is a complete comment:

```
!section 03.02.01(12)D.Taffs 82-04-26
!version 1983
!topic FILE NODE MANAGEMENT

Change component to subcomponent in the last sentence.

!rationale

Otherwise the statement is inconsistent with the defined use of subcomponent in 3.3, which
says that subcomponents are excluded when the term component is used instead of
subcomponent.
```

OFFICE OF THE UNDER SECRETARY OF DEFENSE

WASHINGTON, D.C. 20301

RESEARCH AND ENGINEERING

MEMORANDUM FOR THE COMMANDER, U.S. ARMY MATERIEL DEVELOPMENT AND READINESS COMMAND (DRCDE-SB)
THE CHIEF OF NAVAL MATERIEL (MAT-08Y)
THE COMMANDER, AIR FORCE SYSTEMS COMMAND (ASFC/ALR)

SUBJECT: CAIS Planning Meeting

On 16 November 1983, we held a CAIS Tri-Service planning meeting at the AJPO. Attachment 1 is my Memorandum for the Record of the meeting. Since we all agreed that the establishment of the CAIS was our goal, but that close coordination would be required, I felt that this Memorandum for the Record should be agreed to by the four of us.

Please review Attachment 1 and provide any comments to me by 9 December, so I can revise it. In order to expedite our agreement on the results, I would encourage informal responses to myself as soon as you have them. If you have additional things that you would like to see in it, please provide them also.

Because of the importance of the CAIS to the Ada program, I would like to work directly with you to make sure that we all agree to the results of this meeting.

Robert F. Mathis

Robert F. Mathis
Director
Ada Joint Program Office

cc: HQUSAF/RDST/RDPV
NOSC (832)

MEMORANDUM FOR THE RECORD

DATE: November 16, 1983

FROM: Robert Mathis

SUBJECT: CAIS TRI-SERVICE PLANNING MEETING

ATTENDEES: Dr. Robert Mathis -- Director, AJPO
COL Kenneth Nidiffer -- AFSC, AF Ada PM
COL Harold Archibald -- DARCOM, Army Ada PM
CAPT Dave Boslaugh -- NAVMAT, Navy Ada PM
LTCOL Vance Mall -- Director, STARS
Dr. John Kramer -- IDA
Mr. Jim Hess -- DARCOM
Mr. Owen McOmber -- NAVMAT
MAJ Al Kopp -- AFRDST
LCDR Brian Schaar -- Navy Deputy AJPO
CDR Jim Stewart -- PMS-408
Mr. Bob Converse -- PMS-408
Mr. Pete Fonash -- DARCOM
Ms. Carol Morgan -- STARS Navy Deputy
Dr. Sam Redwine -- MITRE
LTCOL Dick Stanley -- STARS Army Deputy
Mr. Bill Wilder -- SofTech

DISCUSSION ITEMS:

1. Objectives and Overview:

The meeting commenced at 0915. Dr. Mathis welcomed the group, went over the meeting objectives, KIT background and objectives, KIT strategy summary, CAISWG composition and the existing CAIS schedule (Attachment 1, slides 1-5). Jack Kramer then briefed the CAIS concerns and CAIS characteristics (Attachment 1, slides 6 and 7).

2. Reaffirmation of Support for the Tri-Service MOA:

Dr. Mathis discussed the history of the MOA which has resulted in the establishment of the KIT under Navy leadership, the KIT's development of the CAIS, the quarterly tri-service status reports on the KIT program, and the fairly strong tri-Service support for the CAIS effort expressed at the Dallas Tri-Service Review. The result of the discussions that followed was a consensus of positive support for the MOA and CAIS effort, but a recognition that there were concerns with the proposed schedule and its impact on the existing and proposed service efforts.

3. CAIS Implementation Concerns:

The Army stated that it cannot change the existing baseline ALS until after its delivery in January 1985. Upon delivery of the "production quality" release of the ALS in January 1985, they would commence upgrading the ALS CAIS equivalent interfaces to conform to the CAIS if there was a tri-service agreed-to CAIS standard (MIL-STD) at that time. The Navy stated that they needed versions of the CAIS in April and November 1984, and a final standard CAIS document in January 1985 if the CAIS was to become a part of the ALS/N RFP and not impact the existing ALS/N procurement strategy. All parties agreed that the January 1985 version of the CAIS should be a MIL-STD.

4. <u>CAIS Schedule Revision</u>

Dr. Mathis then began proposing a revised CAIS development and standardization schedule based on the original schedule, slide 5, and some of the previous discussions. The principal concerns with the schedule centered around close dependencies between the CAIS standardization, the Army ALS development schedule, and the Navy ALS/N RFP requirements.

The original schedule called for only a clean-up of Version 1.1 of the CAIS based on the existing review cycle. In January 1984 the development of Version 2 was to commence using Version 1 and the requirements document as a basis for development. The resulting Version 2 would have become a MIL-STD by January 1986, which was too late for Navy use in their ALS/N RFP. The first agreement was to make Version 1 a MIL-STD in January 1985. That document would include all the technical comments provided by the public review of CAIS Version 1.1, service technical inputs to be provided by 15 December 1983, and at a minimum those Version 1.1 deferred items which are interfaces in the ALS and AIE.

The second agreement was that Version 2 would not be started until January 1985. At that time the Version 1 MIL-STD would become the baseline from which Version 2 would grow. Version 2 would address three principal areas: first, Version 1.1 deferred items which were not resolved in the Version 1 MIL-STD; second, the requirements in the KIT APSE Requirements for Interoperability and Transportability and Design Criteria for the CAIS (R&C) document which were not met in the Version 1 MIL-STD; and finally those items found necessary as a result of the production release of the ALS, the development of the AIE, the ALS/N contract, and other implementations of the CAIS MIL-STD and tool developments using those interfaces.

5. CAIS MIL-STD Approval:

It was recognized that CAIS approval as a MIL-STD will require active participation on the part of the three services. In particular, the December 1984 approval cycle must be carefully prepared so that all technical and political problems have been resolved prior to that time. All services agreed that they must work positively towards establishing the CAIS as a MIL-Standard which would require identification of the appropriate people to work with the KIT chairman to resolve all technical and program issues prior to December 1984. Each service will have to decide if this is one person or different ones addresssing the technical and program issues. As the first step in meeting the January 1985 date, the services agreed to provide technical comments by 15 December 1983. With respect to the ALS, a meeting between the Army ALS manager, the KIT chairman and several of the CAISWG members was proposed for some time between 12 and 22 December at Ft Monmouth, NJ. This meeting woud permit detailed technical comments and concerns to be discussed so that a plan for their resolution could be executed in January 1984.

6. <u>Configuration Control</u>:

During the discussions of the CAIS schedule it was decided that a close Configuration Management (CM) of the ALS and coordination with the CAIS development was necessary. After a brief discussion, there was a general agreement that the scope of the configuration management should be expanded to include the AIE and the CAIS at some stage, under the same CM group. The actual mechanics for such a CM group were not decided. The Navy suggested that there were existing regulations on how to conduct such a group, and that these should be followed. There was some concern that the existing ALS CM joint-service team may not have a proper charter or be composed of the appropriate personnel to speak for the services.

7. <u>CAIS Schedule Agreement</u>:

Attachment 1, Slide 8 is the final schedule. All parties agreed that it meets the needs of the Army, Navy and Air Force, while proceeding forcefully towards a CAIS MIL-STD.

8. <u>ACTION ITEMS</u>

a. Service PMs will provide a point(s) of contact to work with the KIT chairman.

b. Service PMs agreed to start planning on how they will meet the January 1985 approval of CAIS as a MIL-STD.

c. The Navy agreed to include the 1985 CAIS MIL-STD as part of the ALS/N procurement.

d. The Army agreed to start implementation of the CAIS MIL-STD in 1985.

e. All parties agreed to support a Joint Service configuration management organization.

MEETING OBJECTIVES

16 NOVEMBER 1983

- REAFFIRM TRI-SERVICE MOA
- SUMMARIZE PROGRESS TO DATE
- IDENTIFY ISSUES
- DEVELOP PLANS TO RESOLVE ISSUES

KIT BACKGROUND AND OBJECTIVES

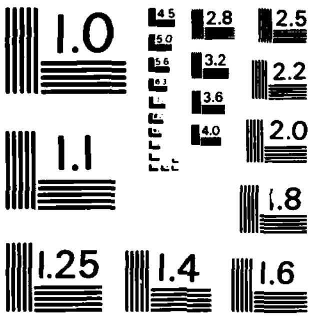
- FORMED AS A RESULT OF THE JANUARY 1982 MEMORANDUM OF AGREEMENT
- A NAVY LED DOD TEAM
- AUGMENTED BY THE KAPSE INTERFACE TEAM FROM INDUSTRY AND ACADEMIA (KITIA)
- CHARTERED TO FORMULATE INTERFACE STANDARDS

  -- TO FACILITATE MOVEMENT OF TOOLS AND DATA BETWEEN APSES

  -- TOWARD WHICH AIE, ALS AND ALL OTHER DOD APSES CAN EVOLVE

AD-A147 648 KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT VOLUME 4(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 30 APR 84 3/5

UNCLASSIFIED NOSC/TD-552-VOL-4 F/G 9/2 NL


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

STRATEGY SUMMARY

- ONE STANDARD SET
- FOUNDATION IN AIE/ALS
- INCREMENTAL DEVELOPMENT
- VALIDATION CAPABILITY
- DOD MAINTENANCE
- EVOLUTION
- TRANSITION

CAISWG COMPOSITION

SEPTEMBER 1982 - ORIGINAL MEMBERS

FOIDL	-- TRW
KRAMER	-- AJPO
OBERNDORF	-- NOSC
TAFT	-- INTERMETRICS
THALL	-- SOFTECH
WIDLER	-- SOFTECH

FEBRUARY 1983 - CAISWG AUGMENTED BY

*FISCHER	-- LITTON DATA SYSTEMS
HARRISON	-- TEXAS INSTRUMENTS
*LAMB	-- BELL LABS
*LYONS	-- SOFTWARE SCIENCES LTD, UK
*MCGONAGLE	-- GENERAL ELECTRIC
*MORSE	-- FREY FEDERAL SYSTEMS
MYERS	-- NOSC
*PLOEDEREDER	-- IABG, W. GER
SCHAAR	-- AJPO
*WILLMAN	-- RAYTHEON
*YELOWITZ	-- FORD AEROSPACE

(*VOLUNTEER KITIA MEMBERS)

CAIS SCHEDULE

16 NOVEMBER 1983 (ORIGINAL)

● DRAFT VERSION 1.0	-- 26 AUGUST 1983
● PUBLIC REVIEW	-- 14-15 SEPTEMBER 1983
● DRAFT VERSION 1.1	-- 30 SEPTEMBER 1983
● PUBLIC COMMENTS	-- 1 NOVEMBER 1983
● SERVICE COMMENTS	-- 15 DECEMBER 1983
● INITIATE VERSION 2.0	-- JANUARY 1984
● DRAFT VERSION 1.2	-- APRIL 1984
● DRAFT VERSION 1.3 (MIL-STD)	-- JULY 1984
● DRAFT VERSION 2.0	-- JANUARY 1985
● FINAL VERSION 2.1 (MIL-STD)	-- DECEMBER 1985

CAIS CONCERNS

- TRI-SERVICE INVOLVEMENT
- PUBLIC INVOLVEMENT
- IMPLEMENTATION INDEPENDENCE
- EVOLUTION
- ALS/AIE CONSIDERATIONS

CAIS CHARACTERISTICS

- HAS A RELATIVELY SIMPLE, UNIFORM UNDERLYING MODEL
- DOES NOT UNDULY INTERFERE WITH IMPLEMENTATION STRATEGIES
- INTEGRATES SMOOTHLY WITH THE FEATURES OF ADA
- PROVIDES A FLEXIBLE FOUNDATION FOR CONFIGURATION MANAGEMENT
- MERGES MODERN OPERATING SYSTEM AND DATABASE MANAGEMENT SYSTEM IDEAS

CAIS SCHEDULE

16 NOVEMBER 1983 (REVISED)

● DRAFT VERSION 1.0	-- 26 AUGUST 1983
● PUBLIC REVIEW	-- 14-15 SEPTEMBER 1983
● DRAFT VERSION 1.1	-- 30 SEPTEMBER 1983
● PUBLIC COMMENTS	-- 1 NOVEMBER 1983
● SERVICE TECHNICAL COMMENTS	-- 15 DECEMBER 1983
● DRAFT VERSION 1.2	-- APRIL 1984
● DRAFT VERSION 1.3	-- NOVEMBER 1984
● MIL-STD VERSION 1	-- JANUARY 1985
● INITIATE MIL-STD VERSION 2	-- JANUARY 1985
● DRAFT VERSION 2 (SEE NOTE)	-- JANUARY 1986
● MIL-STD VERSION 2	-- JANUARY 1987

NOTE: CONFIGURATION CONTROL OF ALS, AIE AND CAIS BY SOME JOINT SERVICE CONFIGURATION CONTROL BOARD

RAC INTRODUCTION

April 1984

Attached following is the version of "DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)" document (informally known as the "RAC") approved at the April KIT/KITIA meeting for public release. A large majority (over 70% in each case) of KIT/KITIA members voted approval, on a section-by-section vote, of the following resolution:

"RESOLVED: This section of the RAC substantially represents the correct set of requirements and design criteria for the CAIS, and shall be placed under configuration management and published in the Public Report."

Two things need to be carefully explained concerning this approval of the draft document:

First, two important sections (4 - "Object Management Support", and 6 - "I/O and Device Control Facilities") are TBD in this version, meaning that no first baseline version of either section has been submitted for KIT/KITIA vote.

Second, many KIT/KITIA members submitted non-major comments with their positive votes (the positive vote was actually titled "Approve with comments"). Many editorial changes (mainly word-smithing) will be made to virtually every paragraph of the document. Additionally, ensuing working group discussions at the April meeting have already led to the formulation of significant rewrites of sections 5 ("Process Facilities") and 7 ("Interoperability and Transportability") as summarized below:

5: Renamed "Program Execution Facilities"; definitions added for undefined terms; significant reorganization of section without major change to conceptual requirements. Additionally, there is ongoing a re-examination of the concept of "process", especially in the areas of data transfer and synchronization between processes, and the level at which those requirements should be stated.

7: Elimination by merging all requirements into other sections with related requirements and criteria.

Therefore, although this draft document is now a configuration-controlled product of the KIT/KITIA and changes to approved sections require formal votes of KIT/KITIA, it is certain that such votes will occur at meetings in July and October of 1984. Initial approvals of sections 4 and 6 are also anticipated at these meetings. KIT/KITIA regard it as a normal process that this document will continue to slowly evolve for a period of years.

In summary, the reader should appreciate that the sense of KIT/KITIA was to expeditiously make available for public examination a version of CAIS requirements and design criteria which confidently represents major directions of KIT/KITIA thinking, despite the fact that at this point in time the document is volatile on many lower level details. Public reviewers who desire to submit detailed critiques of this document are encouraged to contact the KIT chairman or the support contractor for the latest approved version of the document.

Prepared by: Hal Hart (TRW), Chairman
Requirements and Criteria Working Group (RACWG)

Approved by: Patricia Oberndorf (NOSC), Chairman
KAPSE Interface Team (KIT)

W O R K I N G   P A P E R

N O T   A P P R O V E D

DoD Requirements and Design Criteria for the Common APSE Interface Set

23-Mar-1984

THIS IS AN UNAPPROVED DRAFT AND SUBJECT TO CHANGE.
DO NOT SPECIFY OR CLAIM CONFORMANCE TO THIS DOCUMENT.

[THIS VERSION IS FOR DISTRIBUTION TO KIT & KITIA ONLY.]

TABLE OF CONTENTS

PREFACE

The KAPSE Interface Team (KIT), and its companion Industry-Academia team (KITIA), were formed by a Memorandum of Agreement (MOA) signed by the three services and the Undersecretary of Defense in January, 1982. Their purpose is to contribute to the achievement of Interoperability of applications databases and Transportability of software development tools ("I&T"). These are important economic objectives, identified at the outset of the DoD common language initiative in the mid-1970's and now acknowledged to require an integrated Ada Programming Support Environment (APSE), in addition to the standard language Ada, for fulfillment. The core of the KIT/KITIA strategy to fulfill I&T objectives is to define a standard set of Ada Programming Support Environment (APSE) interfaces ("CAIS" for "Common APSE Interface Set") to which all Ada-related tools can be written, thus assuring the ability to share tools and databases between conforming Ada Programming Support Environments (APSEs). Note that a large number of these interfaces are at the Kernel APSE (KAPSE) level. This document establishes requirements and design objectives (called "criteria") on the definition of a CAIS.

This document is related to the DoD "Stoneman" Requirements for Ada Programming Support Environments in identifying and refining the derived requirements which are imposed upon a CAIS and which effect the I&T-related objectives. Additional influences on this document were the DoD "Steelman" Requirements for High Order Computer Programming Languages and the several sets of ANSI "OSCRL" requirements and design objectives for Operating System Command and Response Languages.

This version of the document is NOT APPROVED and is circulated for review and comment; it has been partially reviewed by KIT and KITIA, but is currently incomplete and has not received final approval. Please address all comments in duplicate via ARPANET to "HalHart@ECLB" and "TRWKIT@ECLB.

A form for grading this document, paragraph by paragraph, and according to given statements of merit, is contained in <halhart>RAC.EVAL at ECLB.

1. INTRODUCTION

1A. Scope. This document provides the Department of Defense's requirements and design criteria for the definition and specification of a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs).

1B. Terminology. The precise and consistent use of terms has been attempted throughout the document.

Many potentially ambiguous terms have been used in the document. Most are defined in the Glossary of KIT/KITIA terminology.

Additionally, the following verbs and verb phrases have been used consistently throughout the document to indicate where and to what degree individual constraints apply. Any sentence not containing one of the following verbs or verb phrases is a definition, explanation or comment.

"shall"	indicates a requirement on the definition of the CAIS; sometimes "shall" is followed by "provide" or "support," in which cases the following two definitions supercede this one.
"shall provide"	indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities.
"shall support"	indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities or for CAIS definers to demonstrate that the capability may be constructed from CAIS interfaces.
"should"	indicates a desired goal but one for which there is no objective test.

1C. Relationship to CAIS Specifications and Implementations. This document specifies functional capabilities which are to be provided in the semantics of a CAIS specification and are therefore to be provided by conforming CAIS implementations. In general, the specifications of software fulfilling those capabilities (and decisions about including or not including CAIS interfaces for certain capabilities as suggested by the "shall support" definition in the previous section) are delegated to the CAIS definers. If a particular facility specified in the CAIS is independent of other CAIS facilities, then a CAIS implementor may elect to reuse CAIS facilities to provide the particular specified facility, thereby achieving a "layered implementation" of the CAIS. Therefore, the realization of a specific CAIS implementation is the result of intentionally divided decision-making authority among 1) this requirements document, 2) CAIS definers, and 3) CAIS implementors.

2. GENERAL DESIGN OBJECTIVES

2A. Scope of the CAIS. The CAIS shall consist of the interfaces necessary and sufficient to support the use of APSEs throughout the lifecycle, and to promote I&T among APSEs. The CAIS should be broad enough to support wide sets of tools and classes of projects. The CAIS is not required to provide all general operating system capabilities.

2B. Basic Services. The CAIS should provide simple-to-use mechanisms for achieving common, simple actions. Features which support less frequently used tool needs should be given secondary consideration.

2C. Implementability. The CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a variety of operating systems. The CAIS shall contain only interfaces which provide facilities which have been demonstrated in existing operating systems, kernels, or command processors. CAIS features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for unneeded generality, and to ensure that unused portions of a CAIS implementation will not add to execution costs of a non-using tool. The measures of the efficiency criterion are, primarily, minimum interactive response time for APSE tools and, secondarily, consumption of tool-chargeable resources.

2D. Modularity. Interfaces should be designed in a modular fashion such that they may be understood in isolation and such that there are no hidden interactions between interfaces. This permits a tool writer to employ a subset of the CAIS.

2E. Extensibility. The design of the CAIS should facilitate development and use of portable extensions of the CAIS; i.e., CAIS interfaces should be reusable so that they can be combined to create new interfaces and facilities which are also portable.

2F. Technology Compatibility. The CAIS shall adopt existing standards where applicable. For example, recognized standards for device characteristics are provided by ANSI, ISO, IEEE, and DoD.

2G. Consistency. TThe design of the CAIS should minimize the number of underlying concepts. It should have few special cases and should consist of features that are individually simple. These objectives are not to be pursued to the extreme of providing inconvenient mechanisms for the expression of some common, reasonable actions.

2H. Security. The CAIS shall be implementable as a secure system that fulfills the requirements for a Class (B2) system in the DoD document titled "Trusted Computer System Evaluation Criteria." The CAIS shall be designed to mediate all tool access to underlying system services (i.e., no "by-passing" the conforming CAIS implementation is necessary to implement any APSE function). The CAIS should accommodate implementations that coexist with (without compromising) and operate within a variety of security mechanisms.

3. GENERAL SYNTAX AND SEMANTICS

3.1. Syntax

3.1A. General Syntax. The syntax of the CAIS shall be expressed as Ada package specifications. The syntax of the CAIS shall conform to the character set as defined by the Ada standard (section 2.1 of ANSI/MIL-STD-1815A).

3.1B. Uniformity. The CAIS should employ uniform syntactic conventions and should not provide several notations for the same concept. CAIS syntax issues (including, at least, limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be resolved in a uniform and integrated manner for the whole CAIS.

3.1C. Name Selection. The CAIS should avoid coining new words (literals or identifiers) and should avoid using words in an unconventional sense. Ada identifiers (names) defined by the CAIS should be natural language words or industry accepted terms whenever possible. The CAIS should define Ada identifiers which are visually distinct and not easily confused (including, at least, that the CAIS should avoid defining two Ada identifiers that are only a 2-character transposition away from being identical). The CAIS should use the same name everywhere in the interface set, and not its possible synonyms, when the same meaning is intended.

3.1D. Pragmatics. The CAIS should impose only those restrictive rules, constraints, or anomalies required to achieve I&T. The CAIS specification shall enumerate all instances of syntactic constraint setting which are deferred to the implementor. CAIS implementors will be required to provide the complete specifications of all syntactic restrictions imposed by their CAIS implementations.

3.2. Semantics

3.2A. General Semantics. The CAIS shall be completely and unambiguously defined. The specification of semantics should be both precise and understandable. The semantic specification of each CAIS interface shall include precise statement of assumption (including execution-time preconditions for calls)s, effects on global data and packages, and interactions with other interfaces.

3.2B. Responses. The CAIS shall provide standard responses for all interfaces, including a unique, non-null response (return value or exception) for each type of unsuccessful completion. All responses returned across CAIS interfaces shall be defined in an implementation-independent manner. Everytime a CAIS interfaces is called under the same circumstances, it should return the same response.

3.2C. Exceptions. All named exceptions raised and propagated by the CAIS shall be documented. The CAIS specification shall require CAIS implementations to provide handlers for all unnamed exceptions raised in the implementations' bodies.

3.2D. Consistency. The description of CAIS semantics should use the same word or phrase everywhere, and not its possible synonyms, when the same meaning is intended.

3.2E. Cohesiveness. Each CAIS interface should provide only one function.

3.2F. Pragmatics. The CAIS specification shall enumerate all aspects of the meanings of CAIS interfaces and facilities which must be defined by CAIS implementors. CAIS implementors will be required to provide the complete specifications for these implementation-defined semantics.

4. OBJECT MANAGEMENT SUPPORT

TBD

5. PROCESS FACILITIES

Introduction: Informal definitions of "process," "parent," "child," "dependent object," etc. ... TBD

5.1. Program Invocation and Control

5.1A. Program Invocation. The CAIS shall provide facilities for a process to invoke any other program which access controls allow. Such an invocation places the indicated program into execution creating a child process of the invoking process. Thus a process is defined to be an individual execution of a program. The invoking process is termed the parent process. The CAIS shall provide facilities so that any process may have several child processes executing concurrently. The CAIS shall support the transfer, with possible transformation, of object references as part of process invocation such that the reference received designates the same object in the child process as the original reference designated in the sender. The CAIS shall provide a means for uniquely identifying child processes.

5.1B. Privileges. The CAIS shall be designed such that program invocation and control facilities are not specially privileged or constrained.

5.1C. I/O Redirection. The CAIS shall provide facilities for a tool to redirect process input and output dynamically at process invocation.

5.1D. Wait Options. The CAIS shall provide the following wait options when a child process is invoked: the Ada task enclosing the child process invocation waits for the completion of the child process, and it does not.

5.1E. Resource Control. The CAIS shall provide facilities to allocate, deallocate, and share resources among processes.

5.1F. Interprocess Communication. The CAIS shall provide facilities for interprocess communication. The CAIS shall support the transfer, with possible transformation, of object parameters as part of interprocess communication such that the parameter received designates the same object in the receiver as the original parameter designated in the sender.

5.1G. Interrupts. The CAIS shall provide facilities for a process to interrupt a child process.

5.1H. Process Control. The CAIS shall provide facilities for a process to suspend or resume its child process.

5.2. Process Termination

5.2A. Termination. The CAIS shall provide the facility for a process to terminate its child processes or itself.

5.2B. Return Values. The CAIS shall provide facilities for the return of values or status by a process to its parent (or invoking process).

5.2C. Clean-Up. The CAIS shall assure that the termination of a process provides for the release or termination of all dependent objects and resources (including processes).

5.3. Process Monitoring

5.3A. Process Identification. The CAIS shall provide facilities for a tool to determine the hierarchy of program invocation, subject to access controls and security rights to such information. The CAIS shall provide a facility for a process to determine the unique name of another process.

5.3B. Process Status. The CAIS shall provide facilities to determine the status of a process, subject to access controls and security rights for such information. The CAIS shall provide process status with values sufficient to permit processes to determine, before issuing process control calls, if such process control calls will be effective.

5.3C. Monitoring. The CAIS shall provide interfaces which will permit the control interactions and data capture necessary to support transportable debugging, performance monitoring, and interface validation tools.

5.3D. System Parameter Query. The CAIS shall provide facilities for the query of system environment parameters. (See 3.2F.)

5.3E. Error Logging. The CAIS shall provide facilities for the collection and reading, subject to access controls and security rights to such information, of system-generated and program-generated error messages.

6. INPUT/OUTPUT AND DEVICE CONTROL FACILITIES

TBD

7. INTEROPERABILITY AND TRANSPORTABILITY

7A. Project Interoperability. The CAIS should facilitate inter-APSE exchange of data structures such that large projects can be feasibly moved.

7B. Standard External Form. The CAIS shall specify a "standard external form" sufficient to represent and reconstruct contents, attributes, and relationships of selected data objects.

7C. Import/Export. The CAIS shall support import/export of data represented using the "standard external form".

7D. Character Transmission. The CAIS should support the transmission of all 8-bit character codes.

7E. Backup. The CAIS shall support backup and restoration of CAIS-managed objects with their contents, attributes, and relationships.

7F. Communication Failure. The CAIS shall provide facilities to detect and interface with handlers for disruption of communication, such as detectable accidental disconnection of a terminal.

STONEWG STATUS REPORT

Ann Reedy
Planning Research Corporation

The Stoneman Working Group (STONEWG) is a joint working group of the KAPSE Interface Team and the KAPSE Interface Team for Industry and Academia (KIT/KITIA). The purpose of the STONEWG is to identify and investigate issues raised by or pertaining to the Stoneman document, especially those issues which impact the work of the KIT/KITIA. The STONEWG is currently planning to produce two documents: an "annotated" version of the existing Stoneman document; and a successor to Stoneman.

The current Stoneman document is a very general document. It requires clarification and refinement in many areas. In particular, there are many issues implicit in the document which require resolution from the KIT/KITIA point of view. The annotated Stoneman is a short term product designed to identify the issues raised by the Stoneman and to provide a cross-reference for other KIT/KITIA documents such as the Requirements and Criteria document. The annotated Stoneman will contain the resolution of issues made by the KIT/KITIA on those items which are important to interoperability and transportability issues. The annotated Stoneman should evolve out of a Stoneman analysis. The first phase of the Stoneman analysis has been completed. The completion date for the annotated Stoneman will depend on the speed with which issues are resolved. In that sense, the annotated Stoneman will be a working document for the KIT/KITIA.

The existing Stoneman document was issued in February of 1980. Since that time, a number of APSEs have been designed. Several of these have been implemented and people are currently experimenting with their use. Since 1982, the KIT and KITIA have been grappling with the problem of defining a set of KAPSE interfaces and with other problems concering transportability of APSE tools and interoperability of APSE databases. The STONEWG feels that enough has been learned about APSEs in the last four years that the Stoneman document needs substantial updating. The STONEWG has decided to take on the task of writing a successor to Stoneman. The STONEWG is coordinating its efforts with the Joint Services Software Engineering Environment committee (JSSEE) and with the

Evaluation and Validation Team, both groups which have important input for the new document. No date has yet been scheduled for the completion of this document. A rough draft of an outline for the new document will be reviewed by STONEWG at the July KIT/KITIA meeting.

STONEMAN ANALYSIS: PHASE I

Ann Reedy
Planning Research Corporation

I. Introduction

I.A. Background

The goal of the KIT/KITIA is to define a set of standard KAPSE interfaces for the purpose of achieving transportability of APSE tools and interoperability of APSE databases. Since the Stoneman document describes the general architecture and characteristics of an APSE, the Stoneman sets the context within which the work of the KIT/KITIA must be accomplished. Thus, the Stoneman is an important document to the KIT/KITIA. However, since it is a general document, the Stoneman is not sufficiently precise in many areas of importance to the KIT/KITIA. The joint KIT/KITIA Stoneman Working Group (STONEWG) plans to produce an "annotated" Stoneman for KIT/KITIA purposes. This analysis is the first step towards this annotated Stoneman.

The purpose of this analysis is to aid in the identification of items from Stoneman which need clarification or resolution from the KIT/KITIA point of view. With this goal in mind, the material in Stoneman has been reorganized by concept rather than by the original architectural viewpoint. As issues are resolved in the course of the KIT/KITIA work, these decisions can be recorded as annotations to the Stoneman text. Thus, it is hoped that this analysis wil evolve into the annotated Stoneman. This analysis will also be useful in cross-referencing the Stoneman with the Requirements and Criteria document (a forthcoming KIT/KITIA product) and the Common APSE Interface Set.

I.B. Organization

This document has been organized by dividing the material in Stoneman into "requirements," "criteria," and "rationale" and by grouping together statements dealing with common concepts regardless of their original location in the Stoneman. The concepts selected for the major groupings have been taken from Stoneman itself (paragraph 1.C): major overall requirements (Section II), database requirements (Section III), interface requirements (Section IV), and toolset requirements (Section V). In this preliminary analysis, not all sections of Stoneman have been included (e.g., Stoneman section 2.C Strategy for Advancement). However, every effort has been made to include all Stoneman scetions which deal with APSE requirements.

In this document, the term "requirement" is applied, in general, to statements which are testable. The term "criteria" is applied to statements which describe a capability which is desirable but which cannot be directly tested. The term "rationale" is applied to all other statments which include definitions, descriptions of the possible implementations, etc. Note that the term "requirement" may be applied to a Stoneman statement even if it is not written in the classical requirement format and likewise, the terms "rationale" or "criteria" may be applied to Stoneman statements which are written in the ususal requirement format.

The relevant paragraphs or statements from Stoneman have been included in this document whenever possible. However, many statements have been removed from context or modified by removing irrelevant phrases. Further, no effort has been made to reference the original location of the statements except by Stoneman paragraph numbers. Thus, it is advisable to read this analysis with a copy of the original Stoneman available.

II. Major Requirements

II.A. Purpose of an APSE

II.A.1. Requirements: Stoneman 1.B

1.B The purpose of an APSE is to support the deviopment and maintenace of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications.

II.A.1.a Restatements of Requirements : Stoneman 2.B.1, 3.A, 3.D, 3.E

2.B.1 The overall objective of an APSE is to offer cost-effective support to all functions in a project team engaged in the development, maintenance and management of a software project, particularly in the embedded computer system field, throughout the lifetime of the project.

3.A SCOPE: An APSE shall provide a program development and maintenance environment for embedded computer systems projects involving Ada programs, with the intent of improving long-term cost effective software reliability.

3.D LIFE CYCLE SUPPORT: Support shall be provided to projects throughout the software life cycle from requirements and design through implementation to long term maintenance and modifications.

3.E PROJECT TEAM SUPPORT: An APSE shall support all functions required by a project team. These functions include project management control, documentation and recording, and long-term configuration and release control.

II.A.2. Rationale: Stoneman 1.J, 2.A (all), 2.B.2, 2.B.15, 2.B.16, 4.F.2

Stoneman section 2.A describes the current practice and its deficiencies. It also supplies a definition (by example) of what is meant by "life cycle."

2.B.15 A more comprehensive APSE [than a MAPSE] will offer specialized tools to support a wide range of these [project life cycle] activies, possibly including:

1) Requirements Specification
2) Overall System Design
3) Program Design
4) Program Verification
5) Project Management

2.B.16 . . . a comprehensive APSE may encourage, or even enforce, one specific system development methodology.

1.J It is possible to take a broader and more general view of programming environments as embodying and supporting the complete integrated process of program design and evolution. This generality is regarded as beyond the present scope of the Stoneman; however, the aim is that the present document should not exclude a more general view being developed and so it is intended to be "upwards compatible" in all critical areas.

2.B.2 An APSE adopts a host/target apprach to software construction. That is, a program which will execute in an embedded target computer is developed on a host computer which offers extensive support facilities. Except where explicitly stated otherwise, this document refers to an APSE running on a host machine and supporting development of a program for an embedded target machine.

4.F.2 In describing an APSE, it is therefore necessary to specifiy three parameters:

(a) On what host machine does it reside? (e.g., the CDC6600)

(b) What targets does it support in the sense of (a) [down-line testing capability] or (b) [remote testing capability] above? (e.g., PDP/11)

(c) For which further targets does it generate code? (e.g., Intel 8080)

It is also necessary to specify whcih tools are appropriate for use on a target in full or possibly in degraded form.

II.A.3. Criteria: Stoneman 2.B.3, 2.B.7, 3 (all)

2.B.3 An APSE offers a coordinated and complete set of tools which is applicable at all stages of the system life cycle, from initial requirements specification to long-term maintenance and adaptation to changing requirements.

2.B.7 At all stages of the devlopment of a program--design, coding, testing, maintenance--an APSE encourages the programmer to work in Ada source terms, rather than in terms of the assembly inaguage of the particular host or target machine.

Stoneman section 3 forms a useful set of criteria although it also contains some requirements. The following section 3 paragraphs are included here as they are not relevant to other sections of this document.

3.B QUALITY: An APSE shall reflect the priorities for software quality in miliary embedded computer applications; that is reliability, performance, evolution, maintenance and responsiveness to changing requirements.

3.K HARDWARE: An APSE will be designed to exploit, but not demand, modern high capacity and high performance host system hardware.

3.L ROBUSTNESS: An APSE will be a highly robust system that can protect itself from user and system errors, that can recover from unforseen situations and that can provide meaningful diagnostic information to its users.

Note that a definition of "coordinated set of tools" from 2.B.3 may be found in 3.M. The reference to Ada source terms is also found in 3.C.

II.B. Portability

II.B.1. Requirements: Stoneman 1.D

1.D A further goal of great importance in some areas of Ada usage, such as within the DoD, is that of portability both of user programs and of the software tools within the APSE. The Stoneman, therefore, goes on to indicate an approach to portability by giving requirements for two lower levels within the APSE: the Kernel (KAPSE) and the minial toolset (MAPSE).

II.B.1.a Restatements of Requirements : 2.B.10

2.B.10 The above paragraphs outline the facilities offered by an APSE to its users in support of Ada programming. However, a further requirement is for portability both of APSE tools between, for example, APSEs hosted on different machines and of complete APSE toolsets. To address this aim and to indicate a means of implementation of an APSE designed to provide portability, this document gives requirements for a low level portability interface and support function set (the KAPSE) together with a minimal toolset (the MAPSE).

II.B.2. Rationale: none

II.B.3. Criteria: Stoneman 3.H, 3.J

3.H SYSTEM PORTABILITY: An APSE shall be portable so far as practicable. this will normally be achieved by writing the system in Ada, and by folowing the KAPSE design model as required in this document.

3.J PROJECT PORTABILITY: An APSE shall be designed to facilitate the easy movement of project support from one host machine to another.

II.C. APSE Structure

II.C.1. Requirements: Stoneman 1.E

1.E It is convenient to represent an APSE which addresses these problems as a structure with a number of layers or levels:

Level 0: Hardware and host software as appropriate

Level 1: Kernel Ada Program Support Environment (KAPSE), which provides database, communication and run-time support functions to enable the execution of an Ada program (including a MAPSE tool) and which presents a machine-independent portability interface.

Level 2: Minial Ada Program Support Environment (MAPSE), which provides a minial set of tools, written in Ada and supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs.

Level 3: Ada Program Support Environments (APSEs) which are constructed by extensions of the APSE to provide fuller support of particular application or methodologies.

II.C.2. Rationale: Stoneman 1.C, 1.F

1.C The three principal features of an APSE are the data base, the (user and system) interface and the toolset. The data base acts as the central repository for information associated with each project throughout the project life cycle. The interface includes the control language which presents an interface to the user as well as system interfaces to the data base and toolset. The toolset includes tools for program development, maintenance and configuration control supported by an APSE.

1.F (bullseye diagram)

II.C.3. Criteria: Stoneman 3.C

3.C SIMPLICITY: The structure of an APSE shall be based on simple overall concepts which are straightforward to understand and use and few in number. Whenever possible, the concepts of the Ada language will be used in the APSE.

III. "Database" Requirements

III.A. External Functionality

III.A.1. Requirements: Stoneman 2.B.5, 5.C.3

2.B.5 Individual [database] functions supported by the tools in an APSE include:

(1) Creation

. . .

(2) Modification

. . .

(3) Analysis

. . .

(4) Transformation

. . .

(5) Display

. . .

(6) Linking

. . .

(7) Execution

. . .

(8) Maintenance

. . .

5.C.3 The KAPSE shall provide the database access functions that are required by Ada programs within the APSE.

See also Stoneman 6.A.11.

III.A.2. Rationale: Stoneman 2.B.4, 4.A.1, 4.F.1

2.B.4 The tools communicate mainly via the database, which stores all relevant information concerning a project throughout its life cycle. *The database is structured so that relationships between* objects in the database can be maintained, in order that configuration control problems can be resolved.

4.A.1 The database is the central feature of an APSE system. It will act as the repository for all information associated with each project throughout the project life cycle.

4.F.1 INTER-TOOL COMMUNICATION. Note that where necessary, tools will store information in the database for later use by other tools.

See also Stoneman 1.C, 6.B.1, and 6.B.2. The definition of "database object" can be found in Stoneman 4.A.3, 4.B.1, and 5.B.1.

III.A.3. Criteria: Stoneman 3.C, 3.M, 4.A.2, 5.C.3

3.C Tools will be designed where appropriate to . . . commicate through the common database.

3.M INTEGRATED: An APSE shall provide a well-coordinated set of useful tools, with uniform inter-tool interfaces and with communication through a common database which acts as the information source and product repository for all tools.

4.A.2 The database shall offer flexible storage facilities to all APSE tools.

5.C.3 This [provision of database access functions] shall include the provision of the primitive functions necessary to permit the implementation of access control and security mechanisms as appropriate.

III.B. Object Names

III.B.1. Requirements: Stoneman 4.A.3, 5.A.1

5.A.1 Each object in the database shall have a unique name constructed from a sequence of indentifiers. Each component of the name shall conform to the Ada syntax for identifiers. Where an object is a member of a version group, a version qualifier may be appended to the name.

4.A.3 Every object stored in the databse is accessed by the use of its distinct name.

III.B.1.a. Restatements of Requirements: Stoneman 4.B.1, 4.B.2

4.B.1 An object has a name by which it may be uniquely identified in the database, . . .

4.B.2 All objects in the database are uniquely identifiable; . . .

III.B.2. Rationale: Stoneman 4.A.3, 4.B.1

4.A.3 4.B.1 A separately identifiable collection of information in the database is known as an object.

See also Stoneman 2.B.7 . . .an APSE encourages the programmer to work in Ada source terms . . .

III.B.3. Criteria: none

III.C. Object Content

III.C.1. Requirements: Stoneman 4.B.1, 5.A.2, 5.A.9

4.B.1 An object . . . contains information.

5.A.2 The database shall not impose restrictions on the format of information stored in an object.

5.A.9 The database shall allow APSE tools to access . . . the information content of objects . . .

III.C.2. Rationale: Stoneman 4.B.1, 5.B.1

4.B.1 Typically, an object may contain a separately compilable Ada program unit, a fragment of Ada text, a separable definition, a file of test data, a project requirements specification, an aggregation of othe object-names (i.e., a configuration; see below), a documentation file, etc.

5.B.1 An object in the database consists essentially of:

(a) its contents; that is the raw information it contains, and

. . .

In many systems a complete object is represented by . . . storing the *object itself as an unstructured file.*

See also the defintions of object in III.B.2.

III.C.3. Criteria: none

III.D. Object Attributes

III.D.1.1 General Requirements: Stoneman 5.A.4, 5.A.9, 5.B.1

5.A.4 Objects in the database shall have attributes.

5.A.9 The database shall allow APSE tools to access . . . the attributes of objects . . .

5.B.1 The attributes which record history, categorization and access rights are mandatory. The list of possible attributes is open-ended and some APSEs may provide further attributes.

III.D.1.1.a. Restatements of Requirements: Stoneman 4.B.1

4.B.1 An object has . . . attributes . . .

III.D.1.2. Rationale: Stoneman 5.B.1

5.B.1 An object in the database consists essentially of:

. . .

(b) its attributes; that is, meta-information describing the nature of the object, its history, categorization and so on.

In many systems a complete object is represented by associating the attribute information with the directory entry for the object . . .

III.D.1.3. Criteria: none

III.D.2.1. History Attribute Requirements: Stoneman 5.A.5, 6.A.12

5.A.5 Every object shall have a history attribute.

6.A.12 As minimal functions this tool [the configuration manager] will enable interrogation of history attributes and will offer managerial control over the persistence of objects in the database.

See also Configuration Control.

III.D.2.1.a. Restatements of Requirements: Stoneman 4.A.6, 4.B.4

4.A.6 Mechanisms shall be provided in the database whereby all database objects needed to recreate a specified object will continue to be maintained in the database as long as the specified object itself remains in the database.

4.B.4 It is therefore a requirement at the KAPSE level that history attributes be maintained for all objects . . .

III.D.2.2. Rationale: Stoneman 5.A.5

5.A.5 The history attribute records the manner in which the object was produced and all information which was relevant in the production of the object.

III.D.2.3. Criteria: Stoneman 5.A.5

5.A.5 The history attributes shall contain sufficient information to provide a basis for comprehensive configuration control.

III.D.3.1. Categorization Attribute Requirements: Stoneman 5.A.6

5.A.6 Every object shall have a categorization attribute which indicates the category of information contained in the object.

III.D.3.1.a. Restatements of Requirements: Stoneman 5.B.2

5.B.2 Every object has a category attribute.

III.D.3.2. Rationale: Stoneman 5.A.6, 5.B.2

5.A.6 It shall be possible for the categorization attribute to be used in such a way that APSE tools are offered protection against accessing an object in a way that is not meaningful (i.e., incompatible with the format and /or content of the object) but are not prevented from accessing an object in any way that is meaningful.

5.B.2 In general, a tool will only access objects of an appropriate category. Other tools, however, may need to access objects regardless of their category; one such tool is a general copying tool. The requirement on the KAPSE encompasses both of these styles of access. However, the requirement does not dictate the manner in which protection is offered, nor that the protection mechanism must actually be implemented within the KAPSE.

III.D.3.3. Criteria: none

III.D.4.1. Access Rights Attribute Requirements: Stoneman 5.A.7

5.A.7 Every object shall have an attribute which indicates access rights to the object.

See also Access Controls.

III.D.4.2. Rationale: none

III.D.4.3. Criteria: none

III.E. Relationships Between Objects

III.E.1. Requirements: Stoneman 5.A.3, 5.A.9

5.A.3 The database shall permit relationships between objects to be recorded.

5.A.9 The database shall allow APSE tools . . . to traverse the networks formed by relationships between objects.

III.E.1.a. Restatments of Requirements: Stoneman 4.A.3

4.A.3 The database shall permit relationships to be maintained between objects.

III.E.2. Rationale: none

III.E.3. Criteria: none

III.F. Versions

III.F.1. Requirements: Stoneman 4.A.4, 5.A.1

4.A.4 The database shall permit the user to designate several distinct database objects as forming a "version group." The user shall be permitted to designate one object within the group as being the preferred (or default) version. A method of access shall be offered in which an incomplete object name is provided, sufficient to identify explicitly a version group but not one object within that group; with such access, the preferred version is selected. Every object within a version group shall always be accessible by providing the complete object name.

5.A.1 Where an object is a member of a version group, a version qualifier may be appended to the name.

III.F.1.a. Restatements of Requirements: Stoneman 4.A.5

4.A.5 The configuration control facilities shall allow access to the objects in a version group by the use of an incomplete name.

III.F.2. Rationale: Stoneman 4.B.2

4.B.2 VERSIONS: . . . however, a group of objects may exist as related versions which all may meet the same or closely related external specifications and may therefore be regarded as different versions of the same "abstract object." Within such a group, the user may specify that one object is the normal, default or preferred version which is to be used whenever the user does not indicate a specific one. Typically, in many current systems the concept of the most recent version of a "module" plays an important role and it may be that this methodological choice will be made in many APSEs; however, the requirements do not prescribe this approach.

III.F.3. Criteria: none

III.G. Partitions

III.G.1. Requirements: Stoneman 4.A.7

4.A.7 It shall be possible to establish partitions of the information in the database such that, for example, all objects connected with a specific project area can be grouped in a partition. It shall be possible to associate general access controls with partitions.

III.G.2. Rationale: Stoneman 4.B.5

4.B.5 PARTITIONS: The partition level is the highest level grouping in the database. It exists primarily for managerial purposes as a means of applying broad access and bugetary control to large collections of information associated with projects. It can also play a part in implementation strategies designed to improve access to specifically important partitions in large data bases.

III.G.3. Criteria: none

III.H. Configuration Control

III.H.1. Requirements: Stoneman 4.B.4, 6.A.12

4.B.4 It is a requirement of an APSE that configuration control be provided.

. . .

It is therefore a requirement at the KAPSE level that history attributes be maintained for all objects (see 5.A.5 below) as a baisis for a configuration control system. At the MAPSE level a configuration control system is reuqired but not specified in detail.

6.A.12 CONFIGURATION MANAGER: A tool is required to assist in long term configuration control of projects. As minimal functions this tool will enable interrogation of history attributes and will offer managerial control over the persistence of objects in the database.

III.H.1.a. Restatements of Requirements: Stoneman 4.B.3

4.B.3 The system must contain tools to enable the generation, release and subsequent control of a project which exists in multiple congifurations.

III.H.2. Rationale: Stoneman 4.B.3, 4.B.4

4.B.3 CONFIGURATIONS: Different collections of objects in a project may be brought together to form different groupings or "software configurations." The differences arise in response to, for example, differing categories of user requirements or differences between peripheral devices on various target systems. Some configurations are long-lived, such as major system releases, and others may be temporary test-beds for development purposes. The relationships between objects in different configurations are in general complex, partially overlapping and not well-structured. Some configurations are related in time, such as consecutive "releases;" others co-exist in time as separate "models." Note that configurations are themselves objects and may therefore exist in version groups.

. . .

The automatic rederivation of configurations as a result of constituent changes may be a methodological choice in some APSEs.

4.B.4 . . . In general this [configuration control] necessitates recording and preserving sufficient information to establish, for any extant configuration, its precise contituents and all relevant information to support their repair or modification.

. . .

The detailed requirements on configuration control systems are left to some extent open to design choice. One position to take is that no object whatever can be deleted from the database if it is referenced in the history attribute of any other object. This maximises reliability and maintainability and in many application areas, if combined with an effective archiving system, would be the preferable approach.

However, in other areas the requirements may differ and may indicate that indefinite preservation should be the priviledge of specified objects only (see 4.A.6) and objects not so specified, under managerial control, may be purged from the database.

III.H.3. Criteria: Stoneman 4.A.5, 4.B.3, 5.A.5

4.A.5 The database shall support the generation and control of configuration objects; that is, objects which are themselves groupings of other objects in the database. The configuration control facilities shall allow access to the objects in a version group by the use of an incomplete name.

4.B.3 It is generally necessary to be able to determine for any configuration exactly what are the components of that configuration and to be able to reconstruct in detail the history and antecedents of each component.

5.A.5 The history attributes shall contain sufficient information to provide a basis for comprehensive configuration control.

III.I. Access Controls

III.I.1. Requirements: Stoneman 4.A.7, 4.A.9, 4.B.6, 5.A.9

4.A.9 It shall be possible to associate access controls with any object in the database.

4.B.6 In order to meet these requirements, the KAPSE must have knowledge of individual user identification.

5.A.9 Access protection shall be applied to attributes so that attribute consistency is maintained.

4.A.7 It shall be possible to associate general access control with partitions.

III.I.2. Rationale: Stoneman 4.B.6

4.B.6 ACCESS CONTROLS: The access control requirements on APSEs are expected to be highly specific to the applications area and methodology of each APSE. Some areas will require very detailed controls whereas others will require only a few broad classes of protection.
. . .
These may well be handled by the local underlying operating system.

III.I.3. Criteria: Stoneman 4.A.9, 4.B.6, 5.C.3

4.A.9 Such access controls shall be appropriate for the environment in which the particular APSE system is deployed, and shall be commensurate with the requirement that an APSE supports all roles in a project team throughout the lifetime of a project.

4.B.6 The key requirement for a KAPSE is therefore that the primitive protection facilities it offers shall be sufficiently general purpose so that it can provide the basis for any required access control system in the APSEs built on that KAPSE.

5.C.3 This [provision of database access functions by the KAPSE] shall include the provision of the primitive functions necessary to permit the implementation of access control and security mechanisms as appropriate.

III.J. Consistency and Integrity

III.J.1. Requirements: Stoneman 4.A.12, 5.A.8, 5.A.9

4.A.12 The database shall preserve the consistency of the information and relationships it contains.

5.A.8 The database interface shall permit provision of an archiving facility whereby files may be relegated to backing storage media while nevertheless retaining the integrity, consistency, and eventual availability of all information in the database.

5.A.9 Access protection shall be applied to attributes so that attribute consistency is maintained.

III.J.2. Rationale: none

III.J.3. Criteria: Stoneman 5.A.5

5.A.5 Any necessary constraints shall be imposed on database operations so that the validity and consistency of history attributes is ensured.

III.K. Archiving

III.K.1. Requirements: Stoneman 4.A.6, 5.A.8, 6.A.12

5.A.8 The database interface shall permit provision of an archiving facility whereby files may be relegated to backing storage media while nevertheless retaining the integrity, consistency, and eventual availability of all information in the database.

6.A.12 CONFIGURATION MANAGER: A tool is required to assist in long term configuration control of projects. As minimal functions this tool will enable interrogation of history attributes and will offer managerial control over the persistence of objects in the database.

4.A.6 Mechanisms shall be provided in the database whereby all database objects needed to recreate a specified object will continue to be maintained in the database as long as the specified object itself remains in the database.

III.K.2. Rationale: Stoneman 4.A.11

4.A.11 The capabilities of the APSE database system shall be such that the users may work within the APSE to achieve reliable storage of objects, including long-term storage of archived objects.

III.K.3. Criteria: none

III.L. Management Reports

III.L.1. Requirements: Stoneman 4.B.7

4.B.7 . . . the database should contain information enabling at least two classes of [management] reports to be produced:

(a) Progress reporting . . .

(b) Statistical reporting . . .

III.L.2. Rationale: none

III.L.3. Criteria: Stoneman 4.A.10, 4.B.7, 4.E.11

4.A.10 The database shall store information which allows management reports to be generated, as required at the particular APSE system.

4.B.7 . . .

(a) Progress reporting--budgets, schedules, review and implementation dates, responsibilites, error report tracing, etc.

(b) Statis'cal reporting--usage frequencies, system loading, etc.

4.E.11 Such summary data [provided by APSE tools] shall be stored in the APSE database and will be project-dependent in nature.

III.M. I/O

III.M.1. Requirements: Stoneman 5.A.10

5.A.10 It shall be possible for the actual reading and writing of database objects to be performed from within an Ada tool using the standard input/output facicilities of the language, as defined in package INPUT OUTPUT.

III.M.2. Rationale: none

III.M.3. Criteria: none

III.N. Ada Libraries

III.N.1. Requirements: Stoneman 4.A.8

4.A.8 The database shall support the storage of Ada libraries in source form, and may also support a form where the library object has been pre-compiled for the host or a particular target machine. Facilities for determining the availability and functional specification of library objects shall be provided.

III.N.2. Rationale: none

III.N.3. Criteria: none

III.O. Reliability

III.O.1. Requirements: none

III.O.2. Rationale: none

III.O.3. Criteria: Stoneman 4.B.8

4.B.8 RELIABILITY: The degree of reliability required in the database is specific to the individual application area as it depends on the economically justifiable level of back-up required on the equipment for the project.

IV. "Virtual Interface" Requirelments

IV.A. General Requirements

IV.A.1. Requirements: Stoneman 4.C.1, 4.C.2, 4.C.3, 4.C.9

4.C.1 A virtual interface which is independent of any host machine shall be provided for APSE communication.

4.C.2 The virtual interface shall be based on . . . concepts . . . which are few in number.

4.C.3 The virtual interface shall permit the invocation of individual tools from the APSE toolset.

4.C.9 A mechanism for returning to the underlying operating system shall be provided in the APSE.

IV.A.1.a. Restatements of Requirements: Stoneman 4.D.4

4.D.4 . . . the environment must provide a primitive operation which enables the initiation of a program to be carried out.

IV.A.2. Rationale: Stoneman 4.D.4

4.D.4 The requirements of 4.C.3 . . . *may well be* implemented by a command language (or job control language).
. . .
More precisely, this operation permits a data structure (such as a compiler output) to be executed as a program on the host.

See also Stoneman 1.C.

IV.A.3. Criteria: Stoneman 4.C.2

4.C.2 The virtual interface shall be based on simple overall concepts which are straightforward to understand and use . . .

IV.B. User Interface Requirements

IV.B.1. Requirements: Stoneman 2.B.6, 4.C.4, 4.C.5, 4.C.7, 4.C.8, 4.E.6, 5.D.2, 5.D.3

2.B.6 The user interface offered by an APSE is independent of the host machine.

4.C.4 The user may access the virtual interface from a variety of phyisical terminal devices.

4.C.5 The virtual interface shall permit the user to interact with the invoked tool and to exercise control over the tool.

4.C.7 An APSE must prevent access from the user which might affect the integrity of the KAPSE and its facilities.

4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.

4.E.6 The principles for communication between tools and the user shall be . . . uniform throughout an APSE toolset. The uniform principles shall apply to error handling as well as to normal operation of a tool.

5.D.2 CONTROL FUNCTION LIST The KAPSE shall define a fixed set of terminal interface control functions.

5.D.3 CONTROL KEYS: The list of standard control keys used to initiate these [terminal interface control function] interactions is a convention and is parameterised within the KAPSE. The list may be changed if this is necessary to avoid conflict with other local conventions.

IV.B.1.a. Restatments of Requirements: Stoneman 3.G

3.G UNIFORMITY OF PROTOCOL: Communications between users and tools shall be according to uniform protocol conventions.

IV.B.2. Rationale: Stoneman 4.C.9, 4.D.2, 4.D.4, 5.D.1

4.C.9 Initial user connection to the APSE may require use of the host operating system.

4.D.2 USER INTERACTION WITH TOOLS: The degree of interaction possible between a user and a tool depends on the "granularity" of that style of device; for example, interaction from a batch terminal is limited to initiation of the job, provision of data and parameters and notitication of the completion of the job together with its results.

4.D.4 COMMAND LANGUAGES: Given this [program initiation primitive], one possible approach to the implementation of a command language is to use a basic Ada-like language whose facilities, offered by a simple interpreter tool, provide little more than the ability to perform simple editing of command lines and to intiate programs.

The requirement in 4.C.6 indicates that the primitive initiation facility used by the command language will be made available as a library procedure to Ada programs. This will enable the user to construct job control sequences as Ada program texts which initiate other programs. This use may well be subject to some restrictions; for example, to prevent recursive initiation in unsuitable cases.

A more general approach is to regard the user interaction as being expressed entirely within Ada program segments which are executed or interpreted as necessary in the context of relevant points in the APSE database, thus providing a total Ada environment similar, for example, to an Intelisp environment.

In view of this range of possibilities, the detailed choice of command language is left as a design decision for specific APSEs.

5.D.1 INTERACTIVE TERMINAL CONTROL: During execution of a prgram in an interactive system the requirement exists for the user to be able to have various levels of asynchronous interaction with the program, ranging from requrests to terminate irrelelvant output to commands to terminate the entire session. A provisional list of functions required in given below [5.D.2].

IV.B.3. Criteria: Stoneman 3.F, 4.D.1, 4.E.6, 5.C.6, 5.D.2

3.F USER HELPFULNESS: High priority will be given to human engineering requirements in the design. The system shall provide a helpful user interface that is easy to learn and use, with adequate response times for interactive users and turn-around times for batch users.

4.D.1 VARIETY OF CONTROL DEVICES: The virtual interface will in practice, be accessible to the user in so far as practicable from the terminal devices available with a particular system. In general these may be from three categories:
(a) batch terminals
(b) keyboard interactive terminals
(c) *high band-width graphics* interactive terminals

Where meaningful, the same control signals will be accepted by the terminal interface routine from all devices of these types.

4.E.6 The principles for communication between tools and the user shall be simple . . .

5.C.6 The KAPSE shall provide mechanisms where appropriate whereby asynchronous commands issued by a user at an interactive terminal can be applied to the executing tool.

5.D.2 The list [of terminal control functions] should be short, functionally adequate and human engineered to fit the needs of the terminal user. The following list is proposed:
(a) Issue a request to terminate the current function
(b) Issue a request to terminate the current program
(c) Suspend the current program and establish a new invocation of the command language interpreter
(d) Terminate the current command language interpreter invocation and resume the program suspended when the current CLI was invoked
(e) Abort the current progran and return to its invoker
(f) Abort the current progran abd return to the nearest command language interpreter level.

IV.C. Intertool Interface Requirements

IV.C.1. Requirements: Stoneman 3.M, 4.E.2, 4.E.5, 5.C.4, 5.C.5

3.M INTEGRATED: An APSE shall provide a . . . set of . . . tools with uniform inter-tool interfaces . . .

4.E.2 Tools in an APSE shall be . . . composable with other tools in order to carry out more complex functions where appropriate.

4.E.5 Inter-tool communication shall be via the virtual interface.

5.C.4 The KAPSE shall provide a mechanism whereby it shall be possible for one APSE tool to invoke another APSE tool and supply the invoked tool with parameters.

5.C.5 Input/Output support offered by the KAPSE shall be such that package INPUT_OUTPUT can be used by an Ada tool for communication with the control device from which the tool was invoked.

IV.C.1.a. Restatements of Requirements: Stoneman 4.C.6

4.C.6 APSE tools may access the virtual interface; e.g., to invoke other tools.

IV.C.2. Rationale: Stoneman 4.D.3, 4.D.4

4.D.3 ACHIEVABILITY OF COMMAND FUNCTIONS FROM WITHIN PROGRAMS: The requirement of 4.C.6 is fundamental to the composition of tools.

4.D.4 . . . The requirement in 4.C.6 indicates that the primitive initiation facility used by the command language will be made available as a library procedure to Ada programs.

See also Stoneman 4.F.1.

IV.C.3. Criteria: Stoneman 4.E.3, 4.E.5

4.E.3 Tools . . . where possible shall conform to standard interface specifications.

4.E.5 The communications between tools shall be simple and uniform throughout an APSE toolset.

IV.D. KAPSE Interface Requirements

IV.D.1. Requirements: Stoneman 2.B.13, 5.C.2, 5.C.4, 5.E.1, 5.E.3, 5.E.4, 5.E.5, 5.E.6, 5.E.7, 5.E.8, 5.E.9

2.B.13 . . . the external specifications for the KAPSE will be fixed, . . .

5.C.2 The KAPSE shall offer the input/output support facilities that are required by Ada programs within the APSE which use the standard input/output facilities of the language, as defined by package INPUT/OUTPUT.

5.C.4 The KAPSE shall provide a mechanism whereby it shall be possible for one APSE tool to invoke another APSE tool and supply the invoked tool with parameters.

5.E.1 The KAPSE shall implement interface definitions which shall be available to APSE tools. Such interface definitions shall be given in the form of package specifications in the Ada language.

5.E.3 The source representation of a compilation unit shall be as defined in the Ada reference manual.

5.E.4 An abstract syntax definition of a compilation unit shall be specified.

5.E.5 A post syntactic/semantic analysis intermediate language defintion of a compilation unit shall be specified.

5.E.6 An abstract data type definition of an executing Ada program shall be specified. The abstract data type shall offer read/write access to both the code segments and the data spaces of the executing program. When used in conjunction with the full symbol tables (see 5.E.7 below), this abstract data type shall allow the production of source-level debugging tools.

5.E.7 An abstract data type definition of a comprehensive symbol table for a compilation unit shall be specicifed. In addition to the basic symbol declaration entries, the abstract data type shall encompass source line locations, symbol usage, program topology, and other information required by basic analysis, testing, or debugging tools.

5.E.8 An abstract data type definition of a "library file" (Ada Reference Manual, section 10.4) shall be specified. This abstract data type shall allow new compilation units to be added to the library file, and shall allow the relationship between compilation units in the library file to be determined.

5.E.9 When used in combination, the symbol table and library file abstract data types shall permit construction of comprehensive symbol table(s) for the (possibly incomplete) program(s) represented by the contents of the library file.

See also Stoneman 4.C.7 and 4.C.9.

IV.D.1.a. Restatements of Requirements: Stoneman 2.B.12

2.B.12 The declarations which are made visible by the KAPSE are given in one or more Ada package specifications. These specifications will include declarations of the primitive operations that are available to any tool in an APSE. They will also include declarations of abstract data types which will be common to all APSEs, including the data types which feature in the interface specifications for the various stages of compilation and execution of a program.

IV.D.2. Rationale: Stoneman 2.B.11, 2.B.13, 5.C.1, 5.F (all)

2.B.11 The purpose of the KAPSE is to allow portable tools to be produced and to support a basic machine-independent user interface to an APSE, Essentially, the KAPSE is a virtual support environment (or a "virtual machine") for Ada programs, including tools written in Ada.

2.B.13 While the external specifications for the KAPSE will be fixed, the associated bodies may vary from one implementation to another. In general all software above the level of the KAPSE will be written in Ada, but the KAPSE itself will be implemented in Ada or by other techniques, making use of local operating systems, filing systems or database systems as appropriate.

5.C.1 The KAPSE shall offer the basic run-time support facilities that are required by Ada programs that execute within the APSE.

See also Stoneman 1.E.

IV.D.3. Criteria: Stoneman 5.C.6, 5.D.4, 5.E.2

5.C.6 The KAPSE shall provide mechanisms where appropriate whereby asynchronous commands issued by a user at an interactive terminal can be applied to the executing tool.

5.D.4 PROVISION OF FUNCTIONS: In general, the KAPSE will be specified as a series of Ada package definitions making extensive use of the concept of abstract data types. This technique can be used both to provide KAPSE functions as listed in 5.C above and to provide data structure description of interfaces as in 5.E below.

5.E.2 Interface definitions provided by the KAPSE shall encompass
(i) the primitive operations that the KAPSE makes available to APSE tools; these include any operations that may be neccesary to supplement the facilities of package INPUT_OUTPUT (see 5.A.10) in order to allow an APSE tool access to all the functional capabilities of the database,

(ii) the abstract data types (type declarations plus operations) that are required to interface the various stages of compilation; these include the data types that are produced by a compilation stage for later use by analysis, testing, or debeugging tools.

V. Toolset Requirements

V.A. APSE Toolset Requirements

V.A.1.1. General Requirements: Stoneman 4.E.2, 4.E.3, 4.E.5

4.E.3 Tools shall be written in Ada . . .

4.E.2 Tools in an APSE shall be designed to meet clear functional needs and shall be composable with other tools in order to carry out more complex functions where appropriate.

4.E.5 The communication between tools shall be uniform throughout an APSE toolset.

V.A.1.2. Rationale: Stoneman 7.A (all)

Stoneman section 7.A consists of descriptions of suggested APSE tools. These tools include: an Ada program editor, a documentation system. a project control system, a configuration control system, measurement tools, a fault report system, requirement specification tools, design tools, verification tools, translators, and command interpreters.

V.A.1.3. Criteria: Stoneman 3.M, 3.N, 4.E.5, 4.E.7, 4.E.11

3.M An APSE shall provide a well-coordinated set of useful tools . . .

3.N GRANULARITY: Tools will be designed where appropriate to have separable limited function components that are composable, user selectable . . .

4.E.5 The communications between tools shall be simple throughout an APSE toolset.

4.E.7 An APSE toolset shall offer comprehensive "help" facilities to APSE users.

4.E.11 APSE tools shall provide appropriate summary data for management reports and control.

V.A.2.1. Extensibility Requirements: Stoneman 2.B.8, 3.P, 4.E.8

2.B.8 Extension of an APSE toolset requires knowledge only of the particular APSE and of the Ada language. A new tool--for example, an environment simulator--is written within the APSE. This tool can then be installed as part of the APSE and subsequently invoked.

3.P OPEN-ENDED: It [an APSE] shall permit improvements, updates and replacement of tools.

4.E.8 An APSE toolset shall support its own extension with new tools written in Ada.

V.A.2.1.a. Restatements of Requirements: Stoneman 4.E.4

4.E.4 The set of tools in an APSE shall remain open-ended; it shall always be possible to add new tools.

V.A.2.2. Rationale: none

V.A.2.3. Criteria: Stoneman 3.P

3.P An APSE shall facilitate the development and integration of new tools.

V.A.3.1. Compile, Test, and Debug Requirements: Stoneman 4.E.1, 4.E.9, 4.E.10

4.E.1 . . . an APSE shall support the separate compilation features of the [Ada] language.

4.E.9 An APSE toolset shall permit testing and debugging of any Ada program which does not use machine-dependent features of the language. It shall be possible to perform such testing and debugging purely in terms of the Ada source text and Ada language concepts (i.e., without reference to the instruction set or architecture of any machine).

4.E.10 An APSE shall permit testing and debugging of an Ada program executing in any target machine supported by the APSE. It shall be permitted for such a program to use the machine-dependent features of the language.

V.A.3.2. Rationale: Stoneman 4.F.2

4.F.2 TARGET ENVIRONMENTS. As stated in various sections above, the model of program development expressed in the APSE approach is that of a host-target system where the host offers the vast majority of the support facilities. The whole purpose of the APSE, however, is to develop and support target machine programs and, in embedded computer systems in particular, final testing on the target machine is normally essential. The intention is for that testing to be carried out in Ada terms so far as praticable.

Four general styles of target resident testing are envisaged:

(a) Down-line testing via a host-target macihine link with a target test supervisor resident in the target. The APSE is regarded as distributed between the machines, and the target test superisor is part of the APSE.

(b) Remote testing where the target machine is not directly linked to the host but where the target configuration can support APSE-compatible tools to provide a target-resident test environment. The target-resident part of the APSE amy be regarded as linked to the rest of the APSE by batch-style communication.

(c) Isolated testing on the target machine in cases where target configuration limitations or the application environment preclude the availability of APSE-compatible facilities on the target. In these cases, testing methods will continue as at present to be specific to the target and the application.

(d) Direct testing in situations where the target machine is the same as the host machine.

V.A.3.3. Criteria: Stoneman 4.E.1, 4.E.10

4.E.1 The tools in an APSE shall support the development of programs in the Ada language as defined by the Ada reference manual.

4.E.10 The facilities for testing and debugging of target-resident programs should be based upon the equivalent facilities for host-resident programs.

V.B. MAPSE Toolset Requirements

V.B.1. Requirements: Stoneman 2.B.14, 6.A (all)

2.B.14 The minimal APSE (MAPSE) is one which . . . supports its own extension with new tools written in Ada. Hence, the MAPSE is an APSE and must meet the general requirements set down for APSEs.

6.A.1 TEXT EDITOR: A standard text editor shall be provided with facilities suitable for editing general text, including specifications, design and other documents, and source programs. The editor shall provide the following functions: find, alter, insert, delete, format, input, output, move, copy, and substitute.

6.A.2 PRETTYPRINTER: A display tool is required to format and output textual material ranging form documentation to source programs. More specifically, it shall print database objects in legible formats which depend on the object categorization.

6.A.3 TRANSLATOR: A MAPSE will include at least one Ada translator, which translates source Ada programs into target code for the host and at least one target. The detailed requirements on the translator are that it should interface to the KAPSE as specified in section 5E above and thereby be cooperative with the other tools.

6.A.4 LINKERS: Linkers are required in a MAPSE for both the host and target machines. The facilities needed are:

Partial linking of program units in conformance with language specifications

Creation of an executable program from program units (perhaps partially linked) with the following options:
Logical to physical mapping
Overlay management
Omission of unused compilation units
Creation of a single load file
Linkage map including variable types and unit cross references
Elimination of redundant generic code bodies

6.A.5 LOADERS: Off-line and /or down-line loading shall be supported.

6.A.6 SET-USE STATIC ANALYSER: A tool is required to provide a cross-reference map indicating where each data item is changed in value and where it is merely referenced.

6.A.7 CONTROL FLOW STATIC ANALYZER: This tool produces a chart of the program control topology. This will indicate which routines are called from where in the program and may indicate exception scopes and inter-task communication calls.

6.A.8 DYMANIC ANALYSIS TOOL: On systems with an interactive capability this tool shall provide the following functions:
(a) Snap shot;
(b) Break (with facilities to alter values of variable for interactive use);
(c) Trace;
(d) Interface simulator (for dummy program units);
(e) Statement execution monitor.
(f) Timing analysis.

6.A.9 TERMINAL INTERFACE ROUTINES: Corresponding device handlers shall be provided for each variety of terminal device available on a specific configuration. These interface between the terminal and the relevant functions and data structures in the KAPSE.

6.A.10 FILE ADMINISTRATOR: A file transfer and compare facility shall be provided. A standard transfer format shall be adopted and the following facilities provided:
File Comparison
Error Control
File Transmission
Title Transmission
History Attribute Transmission.

6.A.11 COMMAND INTERPRETER: A command interpreter capable of invoking all APSE tools (potentially with parameters) must be provided by a MAPSE. Users communicating with the command interpreter through an interactive device must be provided with:

1) Facilities for editing and inspecting command lines prior to carrying out the coammand.

2) Positive responses to all interactive operations.

The command language accepted by the interpreter must enable the storage of sequences of commands in the database for later execution.

6.A.12 CONFIGURATION MANAGER: A tool is required to assist in long term configuration control of projects. As minial functions this tool will enable interrrogation of history atrribute and wil offer managerial control over the persistence of objects in the database.

V.B.2. Rationale: Stoneman 2.B.14, 2.B.15, 6.A.1, 6.B (all)

2.B.14 The minial APSE (MAPSE) is one which provides a minial but useful Ada programming environment . . .

2.B.15 For many important acitivites during a project life cycle as listed below, the only support offered by the MAPSE consists of general text manipulation facilities.

6.A.1 The location of the text [by the text editor] may be by line number, by context or equivalent.

6.B.1 The most common action in a programming environment is the manipulation of objects. In a complex APSE, much of this manipulation is done automatically by many of its advanced tools. In a MAPSE, however, these advanced tools are missing and much of the manipulation must be done manually. The easiest way to ensure that general objects can be manually manipulated is to provide a general text editor that can manipulate objects at a low level.

6.B.2 Many of the objects stored in the database are not structured in ways that are immediately legible. The information in these objects (as well as their attributes), however, must often be studied by users; hence the need for a prettyprinting too (or set of tools) that transforms database objects into legible and understandable formats. The prettyprinter may perform transformations that include:
1) Binary to ASCII conversion
2) Indentation of Ada programs
3) Formatting of objects that contain linked lists
4) Formatting of history attributes

6.B.3 In order to execute Ada programs (including the MAPSE tools themselves), it is necessary to translate the programs from (high-leve) Ada to a (lower-level) executable representation. It is of little importance from the development standpoint (although it may be important from other standpoints) whether the translator that is provided is a compiler, an interpreter, or any other type of translator.

6.B.4 Debugging and testing Ada programs at a low level representation (machine or assembly language level, for instance) defeats much of the purpose of programming in Ada in the first place. Therefore, these facilities must be provided at the Ada source level in a MAPSE. This may require KAPSE level support.

V.B.3. Criteria: Stoneman 2.B.16, 6.A.1

2.B.16 Clearly, the MAPSE does not emphasize any particular development methodolody at the expense of any other.

V.C. Library Requirements

V.C.1. Requirements: Stoneman 2.B.9, 4.A.8, 4.E.1, 6.C.2, 6.C.3, 6.C.4

2.B.9 An APSE supports the use of libraries of standard routines for incorporation in programs written for both host and target machines.

4.A.8 The database shall support the storage of Ada libraries in source form . . .

4.E.1 . . . an APSE shall support the separate compilation features of the [Ada] language.

6.C.2 A high-level I/O package is required [in a MAPSE] for each target machine.

6.C.3 A physical file handling package is required [in a MAPSE] for each target, if appropriate.

6.C.4 A file directory system is required [in a MAPSE] for each target.

V.C.2. *Rationale: Stoneman 4.A.8, 6.C.1, 6.D (all), 7.B (all)*

4.A.8 An APSE . . . may also support a form [of Ada library] where the library object hasbeen pre-compiled for the host or a particular target machine.

6.C.1 One or more high level I/O packages are required for the host, to extend or to provide alternatives for the package specified in the language manual. Such packages will provide calling conventions and implementations for standard device handling routines. Both host and target resident packages are required as appropriate.

6.D.1 The requirements in 6.C.3 and 4 are not very specific as the nature of the target environment is project-dependent. However, some file handling package is required on the target to implement the access of target files by the host and the exchange of files with the host. These requirements are expressed separately; it may be on a specific target that either the local operating system file directory structure, or physical file handlers, or both will be utilized to meet the requirements.

6.D.2 Clearly these proposals for MAPSE libraries represent a minimal subset only.

7.B.1 It is expected that the standard libraries provided with APSEs will be oriented towards different application areas and /or methodologies as addressed by each APSE.

7.B.2 As one initial example, there will be a requirement for a numeric applications library.

V.C.3. Criteria: 4.E.1

4.E.1 The tools in an APSE shall support the development of programs in the Ada language as defined by the Ada reference manual.

```
----------------------------------
--                              --
--                              --
--     P R E L I M I N A R Y    --
--                              --
--                              --
----------------------------------
```

Outline for

A USER'S GUIDE TO ADA TRANSPORTABILITY

(Updated for April 1984 meeting of GACWG)

```
----------------------------------
--                              --
--                              --
--     P R E L I M I N A R Y    --
--                              --
--                              --
----------------------------------
```

by Ron Johnson

4/9/84

Table of Contents

Chapter 1 INTRODUCTION

Chapter 4 Ada LANGUAGE CONSIDERATIONS

4.1 Representation Specifications

- How they assist transportability
- How they hinder transportability
- Locating them in programs
- Effects on efficiency
- Alternatives

4.2 Pragmas

- Language-defined pragmas
- Implementation-defined pragmas
- Locating pragmas in programs
- Alternatives

4.3 Use of Standard Types

- Implementation-defined
- Possible problems
- Locating usage in programs
- Defining portable types
- Fixed point definitions
- Array definitions

4.4 Efficiency Considerations

- Differences in computers and APSE's
- Differences in optimization
- Measuring efficiency
- Program modification
- Run-time checks

4.5 Attributes

- Usefulness in developing portable programs
- Relationship to Package SYSTEM

4.6 Unchecked Programming

- Dangers
- Locating unchecked programming in programs

4.7 Package STANDARD

- Care with type definitions
- Locating use of particular types
- Package SYSTEM

4.8 Parameter Passing Issues

4.9 Use of Ada I/O

4.10 Various Pragmatics (see Herb Willman's paper in NOSC TD 509)

4.11 Appendix F of Ada LRM : A Discussion

Chapter 5 APSE CONSIDERATIONS

Chapter 6   H A R D W A R E   C O N S I D E R A T I O N S

```
-- Much of the following list of topics was obtained
-- from Reference [4].  Some topics are germane to
-- this document, while others may not be;  all should
-- probably be given some consideration.  As an approach,
-- any relevent topic could be discussed as follows:
--
--          6.3  Internal Clock
--             6.3.1 Statement of Problem
--             6.3.2 Implications
--             6.3.3 Solution
--             6.3.4 Associated Transport Aids
--
--
-- Topics for consideration include:
--
--        . Character set differences (internal, tape, card,
--          and collating sequence)
--
--        . Internal clock
--
--        . Computer speed
--
--        . Memory size
--
--        . Byte ordering
--
--        . Peripherals (number and type)
--
--        . Sense devices (swithes, lights)
--
--        . Printer speed and line size
--
--        . Number representations (negative/positive,
--          floating point, sizes)
--
--        . Internal vs external number forms
--
--        . Interrupts (types, mechanisms, priorities)
--
--        . Boundary allignment
--
--        . Rounding
--
--        . Architectures (micros, pipeline, array processor,
--          virtual system, Ada computer)
--
--        . Sizes (words, characters)
--
--        . Input/Output (operations, data formats, unique
--          external devices)
--
--        . Memory management
--
--        . Terminal types
```

Chapter 7 PROCEDURAL CONSIDERATIONS

7.1 Transport Media

- Cards
- Tape
- Disk
- Data Link

7.2 Data Transfer

- Files
- Data Bases
- Node Structures
- CAIS Considerations
- Tools which assist
- Networking

7.3 Program Representation Transfer

- Source Representation
- DIANA (or equivalent)
- Other

7.4 Levels of Software Transfer

- Individual Program or Tool
- Tool Set
- System

7.5 Documentation Requirements Reflecting Software Port

- Modifications required because of program change
- Porting the documents themselves
- Tools for assistance

7.6 Configuration Control Issues

7.7 Modifying Ported Software

7.8 Testing Ported Software

7.9 When and How to Re-Write Code

7.10 Documentation Requirements Unique to Portable Software

7.11 External Code of Data Files

Chapter 8 S T Y L E C O N S I D E R A T I O N S

```
-- Our objective is NOT to write a style guide.  However,
-- we must discuss style issues that bear on IT.  And
-- if we discuss ONLY IT-related style issues, then how
-- do we keep them in context ???
--
-- Thus I propose that we DO write a short style guide,
-- by presenting A STYLE (not claiming it to be the
-- ULTIMATE STYLE) and by embedding our IT-related discussion
-- within the context of that style.  We might derive this
-- style by examining References [6], [7], [8], and others.
--
-- The IT-related things might include:
--
--      . Modularity
--
--      . Isolation of system-dependent code and data
--
--      . Visibility of system-dependent code and data
--
--      . Naming conventions (for visibility and automation)
--
--      . Documentation conventions
--
--      . Use of Ada as PDL
--
--      . Use of other design expressions
--
--      . Typing
--
--      . Use of comments
--
--      . Development of portable version before system-
--        dependent version
--
--
--      others  T B D
--
```

CHARTER of the COMPLIANCE WORKING GROUP of the KIT/KITIA

Scope

The compliance working group of the KIT and KITIA has been formed to address topics in two major areas. Stated in very general terms, the first is the adherence of KIT/KITIA products to any stated or written set of objectives. This includes the adherence of the CAIS to the Requirements and Criteria document and probably other things such as the adherence of the Requirements and Criteria document to the stated objectives of I and T, or CAIS adherence to ANSI or other standards. The second topic that we will address is the adherence of any implementation, design, or whatever to any products generated by the KIT/KITIA. While this is a general statement that says we will be addressing the technology needed to measure adherence of a CAIS implementation to specifications, it also indicates that we will be concerned with alternative designs of the CAIS and the way in which they satisfy the Requirements and Criteria documentt.

The scope of the working group is intended to be sufficiently general so as to allow us to address technical issues related to conformance. That is, we will not specifically be measuring whether a specific implementation of the CAIS adheres to the specifications, but rather are concerned with the issues related to validation of a CAIS. Further, we are concerned with developing techniques related to compliance, and assessment procedures.

The working group considers itself to be primarily a technical body as opposed to addressing policy as it relates to compliance. While a technical slant is an explicit goal and direction, often the results of technical considerations imply policy recommendations.

DRAFT KIT/KITIA GLOSSARY

for words cited in:
RAC 1984 Feb 17
CAIS 1.1 1983 Sep 30

KEY

* - Words followed by * are defined by the CAIS document in terms of the CAIS design.
[C] - Term is cited in the CAIS document.
[R] - Term is cited in the RAC document.
[C/R] - Term is cited in the CAIS and RAC documents.

TERMS

absolute path* [C] - A path starting at a top-level node is called an "absolute" path.

access control [C] - (TCSEC)
[discretionary access control] - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject.
[mandatory access control] - A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e. clearance) of subjects to access information of such sensitivity.

Ada Programming Support Environment (APSE) [C/R] -
(UK Ada Study) - (STONEMAN). The purpose of an APSE is to support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications. The principal features are the database, the interface and the toolset. It is structured in levels:

level 0: hardware and host software as appropriate
level 1: KAPSE
level 2: MAPSE
level 3: APSEs which are constructed by extensions of the MAPSE to provide fuller support for particular applications or methodologies.

attribute* [C/R] - A named value associated with a node or a relationship.

base* [C] - A path can start at a known (not necessarily top-level) node and follow a sequence of relationships to a desired node. This is a "relative" path and the known starting node is called the "base".

canonical name [R] - (RAC) the name unique to an object which is guaranteed by RAC to exist.

content(s) [C] - Data, characters, words, or other units which are held specifically addressable in some storage unit; (CAIS) a distinguished attribute of a node.

criterion [R] - (RAC) design objective.

current process* [C] - The currently executing process making the call to a CAIS operation. It defines the context in which the parameters are to be interpreted.

default relation* [C] - Many CAIS operations allow the user to omit the relation name when referring to a relationship, defaulting it to "DOT". DOT is therefore referred to as the default relation name.

device [C/R] - a piece of equipment or a mechanism designed to serve a special purpose or perform a special function.

device node* [C] - A node whose content represents a logical or physical device.

file [C] (LRM 14.1.1) [Ada external file] - Values input from the external environment of the program, or output to the environment, are considered to occupy external files. An external file can be anything external to the program that can produce a value to be read or receive a value to be written.

file node* [C] - A node whose content is an Ada external file.

interface [C/] (DACS)

(1) A shared boundary.

(2) the set of data passed between two or more programs or segments of programs, and the assumptions made by each program about how the other(s) operate.

(3) The common boundary between software modules, between hardware devices, or between hardware and software modules.

(4) When applied to a module, that set of assumptions made concerning the module by the remaining program or system in which it appears. Modules have control, data, and services interfaces.

An interface can be represented by means of Ada package syntax and semantics.

interoperability [C/R] - (K/K) Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.

KAPSE [R] - (UK Ada Study) That level of an APSE which provides database communication and runtime support functions to enable the execution of an Ada program (including a MAPSE tool) and which presents a machine-independent portability interface.

MAPSE [R] - (UK Ada Study, STONEMAN) That level of an APSE which provides a minimal set of tools, written in Ada and supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs. The term is used in this [UK] study to mean not a strictly minimal set, but a set with which a user can happily work.

node* [C] - representation within the CAIS of an entity relevant to the APSE.

node handle* [C] - a reference to CAIS nodes that is internal to a process.

object [R] - (RAC) separately identifiable collection of data (treated as an undefined term explained by its properties).

path* [C] - A sequence of relationships connecting one node to another.

pathname* [C] - Starting from a given node, a path is followed by traversing a sequence of relationships until the desired node is reached. The "pathname" for this path is made up of the concatenation of the denotations of the traversed relationships in the same order in which they are encountered.

pragmatics [C/R] - [implementation pragmatics] Constraints imposed by an implementation or use that are not defined by the syntax or semantics.

primary path [C] -

primary relationship* [C] - When a node is created, a primary relationship must be initially established from some other node, called its parent node.

process* [C/R] - the execution of an Ada program including all its tasks

process node* [C] - A node whose content represents a CAIS process.

program [R] - (LRM) A program is composed of a number of compilation units, one of which is a subprogram called the main program, which may invoke subprograms declared in the other compilation units of the program.

relation [C] -

relation name* [C] -

relationship* [C/R] -

relationship key* [C] -

secondary relationship* [C] - Secondary relationships are arbitrary connections which may be established between two existing nodes.

security [R] -

source node* [C] - Node from which a relationship emanates.

structural node* [C] - A node without content. It is used to structure other nodes.

target node* [C] - Node to which a relationship leads.

terminal [R] -

tool [C/R] - (IEEE) [software tool] - A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, automated design tool, compiler, test tool, maintenance tool.

top-level node* [C] - A node that has no parent relationship.

transportability [C/R] - (K/K) - Transportability of an APSE tool is ability of the tool to be installed on a different CAIS implementation; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonyms.

virtual terminal [C] - (Davies) a conceptual terminal which is defined as a standard for the purpose of uniform handling of a variety of actual terminals.

user [R] -

REFERENCES

ANDIP - Definition taken from the American National Dictionary for Information Processing, X3/TR-1-77, September 1977, American National Standards Committee, X3 - Computers and Information Processing

CAIS - Definition taken from the Draft Specification of the Common APSE Interface Set (CAIS), Version 1.1, 30 September 1983, KIT/KITIA CAIS Working Group for the Ada Joint Program Office.

DACS - Definition taken from the DACS Glossary, a Bibliography of Software Engineering Terms, GLOS-1, October 1979, Data & Analysis Center for Software.

Davies - Definition taken from Davies, D. W., Barber, D. L. A., Price, W. L., and Solomionides, C. M., "Computer Networks and Their Protocols," John Wiley & Sons, New York (1979)

IEEE - Definition taken from the IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983.

ISO - Definition was identified by the source as developed by Technical Committee 97, (Information Systems), Subcommittee 1, (Vocabulary) of the International Organization for Standardization.

K/K - Definition taken from the KITIA Public Report, Volume I, 1 April 1982.

LRM - Definition taken from the Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, Feb. 17, 1983, United States Department of Defense.

PR#1 - KIT/KITIA Public Report No. 1.

STONEMAN - Requirements for Ada Programming Support Environments "STONEMAN", Department of Defense, Feb. 1980.

TCSEC - Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, CSC-STD-001-83, 15 August 1983.

TDS - Tactical Data Systems Glossary, MAT-09Y, 1976.

UK Ada Study - Definition taken from the United Kingdom Ada Study Final Technical Report, Volume I, London, Department of Industry, 1981.

Ada ENVIRONMENTS AS GFE IS HARMFUL

D. E. Wrege

Control Data Corp.

Introduction

Both the Army and the Navy put forth relatively stringent policies regarding the use of Ada Programming Support Environments (APSEs) at the Dallas AdaTEC meeting. Of the policy statements put forth, the Navy's is not only the most stringent, but also is documented in the Ada Language System/Navy (ALS/N) RFI. Therefore this paper will specifically address the Navy policy as set forth in that document. The intent here is not so much to cast stones at the Navy, but rather to point out why policies similar to those so successful in the past, such as those relating to MTASS, will have a disastrous effect on the Ada Initiative.

Quoting from the System Specification for the ALS/N draft of 15 June 1983:

"3.1.6.6 MAPSE Distribution Concepts

The distribution of the MAPSE category software will be consistent with existing PMS 408 distribution policies as currently implemented for MTASS distribution. The major concepts are:

a. The software will only be delivered to those sites directly involved in the development of embedded systems for the Department of Defense.

b. The user site will not be able to modify the software because neither source text nor source listings will be delivered.

c. All user sites will receive the same software for a specific host computer; the site may tailor the software to project requirements by either deleting delivered tools or adding new project-specific tools."

No Compilers

The DoD has seen to the design and definition of Ada. They have copywrited the name to ensure that Ada is Ada. They have developed a validation facility and encouraged the world community to develop Ada capabilities. There are even a large number of independent efforts underway, that are being underwritten by private funds. And now, suddenly, these developers are being told that the government, and in particular the Navy, is not a marketplace for these compilers. The reason for the independent development of Ada compilers is for use in building systems for the government, or to sell computers to companies that develop systems for the government. If industry is developing compilers only for themselves, they will surely throw out the difficult parts to implement and not attempt validation, thereby not having Ada. The DoD has guaranteed, by the complexity of the language, that the only possible motivation for implementing full Ada is because the DoD requires it for developing software under government contract. But a GFE policy disallows the use of compilers not provided by the contracting agency. Thus, independently developed compilers will exist only to the extent that portions of the government do NOT follow GFE policies like the Navy has put forth.

The above stated Navy policy is clearly contrary to what the AJPO has been attempting to do. why establish an Ada Validation Office (AVO) if developers are forced to use a GFE compiler? The validation suite then would be merely an acceptance test for government sponsored compilers. This was clearly NOT the intent of the AVO when it was established. Actually, if all of the service components had a GFE policy similar to the Navy's, the goals of transportability and interoperability between APSEs makes sense only from the standpoint of sharing tools between government components. It is probably cheaper to just convert the non-redundant tools than to, for example, retrofit both an ALS and an AIE and all of their tools to the CAIS.

No Environments

There are absolutely no incentives for developing independent environments that are compatible with the ALS/N assuming the strict GFE policy stated. [Except perhaps by chance.] The Navy policy disallows even the transporting of ALS/N tools to an independently developed APSE since neither source code nor listings will be made available. [Another good reason to make all of these tools portable.]

But why should the Navy care? This policy has worked well with MTASS. They have rehosted it on almost all reasonable systems. They should be able to do it again with the ALS/N. WRONG! The rehosting of MTASS was possible because the MTASS tools were written in FORTRAN. FORTRAN compilers were readily available on virtually all computer systems. ALS and ALS/N tools are written in Ada and thus cannot simply be transported to a new host. Ada compilers are obviously not widely available and will not likely become so if a strict GFE policy is followed. MTASS tools also depend upon a set of relatively simple Common Interface Routines (CIR) for operating system support. The KAPSE or even the CAIS is far more complex than the CIR. Therefore the Navy must not only spend big bucks to retarget their Ada compiler to the host machine but also may need to modify the host operating system to implement the KAPSE/CAIS.

A Possible Solution

There have been legitimate situations which encouraged the strict GFE policy followed by the Navy. Foremost is the fact that all too often the developer of a system used tools that were required for the life cycle maintenance of the system, and for one reason or another these tools couldn't be run at the maintenance facility. The result was either an undesired dependence on the developer (sole source at any cost) or a premature shortening of the useful life of the system. Even when the tool could be transported to the maintenance facility, that tool whould have to be maintained. Eventually the maintenance organization wpi;d be spending an increasing proportion of its resources maintaining tools rather than weapon systems. One solution, and the one adopted, was for the Navy to provide the tools for development and to require that they be used. In this way the Navy has ownership of the tools and can guarantee that they will be able to maintain the weapons system software after delivery.

An environment is defined by its tools. The project, such as a weapons system, is defined by its data, i.e. sources, binaries, documentation, etc., and the operability of the environment tools on that data. Therefore the system can be transmitted properly to the maintenance activity provided that (1) the database can be moved to the maintainer's APSE, and (2) the APSE toolset of the maintainer can operate on that database in all manners necessary to continue operation and maintenance of the system. Clearly this can be accomplished using a strictly GFE environment. But consider an alternative policy which requires the establishing of a maintenance baseline by the contractor through:

1. moving only the non-derivable database objects. This means all sources, documentation, testing files, etc. but not, for example, binaries that can be derived from the source by using a compiler.
2. reconstructing the system using the tools of the maintenance APSE.
3. retesting the system using the tools of the maintenance APSE if required.

The ability to perform the above successfully would define "compatibility" of a developer's APSE with a maintainer's APSE.

To encourage contractor/vendor investment in environments, and to reduce potential risk in the transmittal of systems, tools used in the maintenance APSE should be made available to those contractor/vendors. These tools should be placed in the public domain, and made as transportable as possible. They should be written in a common language, Ada, and depend on only a standard set of virtual operating system interfaces (e.g. the CAIS).

[But now we are on familiar ground.] In this way it should be fairly easy for contractors to start with a baseline APSE which is compatible with the government APSE. Note that the tools are NOT GFE. It is the contractor/vendor's responsibility to convert and maintain their version of the tools. They can improve on them, but will be deterred from evolving them to the point where they are no longer "compatible" with the maintaining APSE.

KITIA POLICY RECOMMENDATIONS

The KITIA recommends against adoption by any DoD component of a policy requiring contractors to use a specific APSE for development work. We believe such a policy will seriously inhibit independent development of APSEs and Ada tools and is therefore detrimental to the goals of the Ada program.

The KITIA recommends an alternative policy based on transmittal requirements. This policy will both allow the contractor to select the development environment while satisfying the services' needs for a stable operations and maintenance environment. A project is defined by its data and the capability of the environment's tools to operate on that data. A project may be successfully transmitted to the maintenance activity provided that:

1. The project data can be moved to the maintenance APSE

2. The maintenance tool set can operate on that project data as necessary for operation and maintenance of the system

This policy requires such a transmittal of the system to the maintenance APSE by the contractor.

A possible transmittal procedure under this policy is to:

1. Move only the non-derivable project data

2. Reconstruct the system using the maintenance tool set

3. Accept the system

In order to ensure that the movement of the data is feasible, and that reconstruction results in a working system, the tools resident on the maintenance APSE necessary for development must be transportable and available to the contractor. These goals are satisfied by implementing ddevelopment and maintenance tool sets using a common APSE interface set.

SECURITY REQUIREMENTS for Ada PROGRAMMING SUPPORT REQUIREMENTS

David C. Pacterchik

```
*******************************************
*                                         *
*                 DRAFT                   *
*                                         *
*******************************************
```

DRAFT 1/16/84

Abstract

This document addresses security requirements for Ada Programming Support Environments (APSEs) intended to hold classified information, and how the Common APSE Interface Set (CAIS) currently under development might accomodate them. A review is given of the initiatives, instructions, regulations and mandates which require a secure APSE, together with the common-sense rationales which accompany them. Mandatory security requirements and their relationship to classifications and clearances are presented, first generally and then with specific reference to the APSE concept. Finally, the concept of a working classification range is considered, with attention paid to the distinction between requirements for trusted tools and requirements for the underlying Kernel APSE (KAPSE).

*UNIX is a trade/service mark of the Bell System.

TABLE OF OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Concluded)

LIST OF ILLUSTRATIONS

SECTION 1

INTRODUCTION

1.1 Scope of Document

This document addresses the requirements for security in the Ada [1] Programming Support Environment (APSE) intended for use by the Department of Defense.

The requirements which are already in place on Ada and its mandated user community are first reviewed, with some view as to how they tie together. A review of general mandatory security requirements, in the context of an APSE, is then presented. Security requirements on a multi-level secure APSE are then proposed. This is followed by a discussion of security requirements for APSE tools.

1.2 DoD-Wide Initiatives, Instructions and Mandates

1.2.1 The Ada Initiative

The Department of Defense has issued an initiative *[ADAINIT]* intended to reduce the life cycle costs associated with software development, maintenance, and changes in target hardware. This initiative is to be implemented, in part, through the use of a standard high order language, namely Ada.

1.2.2 Mandates Regarding Trust

Mandates exist within the DoD which call for computer-related activities to be conducted in a reliable and non-compromising (which this document will call "trusted" or "secure") fashion where sensitive or critical DoD information is to be processed. Chief among these are the instruction [5200.28], which provides the general DoD ADP security mandates and [5200.1R], which mandates the need for classification management mechanisms (e.g. downgrading, labeling and handling procedures). The requirement for accreditation (the authorization and approval for an ADP system to process sensitive or critical information in an operational environment) may be viewed as the operational "bottom line" among these requirements.

[1]Ada is a registered trademark of the United States Department of Defense Ada Joint Program Office (AJPO).

1.2.3 The APSE Onion Skin

In support of [ADAINIT], Ada encompasses not only a high-order language, but also an Ada Programming Support Environment (APSE). The APSE provides for a wealth of programming tools which support development, documentation, debugging, command-level user-system interface and configuration control. The requirements for the APSE are spelled out in [STONEMAN].

The APSE is designed to be as host-independent as possible. It is conceptually divided into major components which are often portrayed graphically in approximately concentric fashion (hence the descriptive term "onion skin"). It splits roughly into a "derivable" tool set (which may contain application- or target-dependent tools), the minimal toolset portion of the APSE (known as the minimal APSE, or MAPSE) and a kernel APSE (KAPSE). The KAPSE is the not necessarily an operating system (OS), but is the only "host-like" portion of the APSE, interfacing the toolset to the underlying host software and hardware.

1.3 Impacts of the Initiative on Project Sponsors

As the reader should be aware, each organization in the Department of Defense is required to implement [ADAINIT] as a way of meeting its own software development requirements for new and advanced-development programs, particularly those involving embedded systems. The Army has already contracted substantial work on an Ada Language System (ALS) with extensions to support the NEBULA-architecture Military Computer Family (MCF) together with the MCF Operating System (MCF/OS), which comes in several flavors. The Air Force has initiated a similar effort with the Ada Integrated Environment (AIE). The Navy has come somewhat lately to this effort, but has its own requirements for an Ada system, based in large part upon the ALS effort [SOWALSN, SSPECALSN]. In specifying security-specific APSE requirements, this document recognizes that, while [STONEMAN] is currently the ultimate source of general APSE and KAPSE requirements, ALS, [CAIS] and [SSPECALSN] provide an excellent model (actually a "meta-implementation") of an APSE upon which to specify security requirements.

1.4 Holes in the APSE Onion Skin

1.4.1 Consequences of Mandates Regarding Trust

The collection of mandates noted in Section 1.2 above has a number of implications for the way in which DoD conducts its computer-related activities. The DoD Computer Security Center (DoDCSEC) has published "Trusted Computer System Evaluation Criteria" [TCSEC] which address the question of determining the degree of "trustedness" of a computer system product. Accreditation is meant as a statement of whether or not a system is authorized to operate in a given security mode on the information it processes (a pass-fail decision). Evaluation is a statement of the degree (literally a letter grading system, with refinements) of trust to enforce a given policy or scheme which may be placed in that system. Evaluation may be performed in abstracto; accreditation must (by mandate of [5200.28]) be performed in situ.

1.4.2 The APSE as a Genie in a Bottle

Trust must begin somewhere. The security model of Bell and LaPadula [BELLAP], which is a basis for many of today's computer security models and policies, has the "start trusted and stay trusted" concept at its heart. Most trusted computing systems only ensure the "incremental preservation" of trust, i.e., none of their actions alter an existing state of trust. Such systems are trusted not under arbitrary conditions, but rather when given that trust exists initially. Although this notion applies at the instruction cycle level, its common sense is relevant even at the (much larger) software life cycle level: the sooner a system begins operating in a trusted mode, the less likely compromise is to occur.

Unfortunately, it is a common assumption that software development systems (which include not just host hardware, but also the software tools running on the hardware, the physical facility and all the development personnel as well) are somehow inherently benign. An effective software development system is often viewed as a trusted servant. It may be likened to a bottled Genie, that may be invoked to benignly do a client's bidding (develop the software that client needs) at the mere flick of a cork. This is a dangerous assumption: trust does not begin here. It cannot be assumed that the Genie and all its potential invokers are without evil intent. The recent, real-life penetration of various government facilities in "War Games" style is concrete proof of this.

1.4.3 How Dangerous is the Genie?

These comments are directed at the concept of a "system" as encompassing not only the target environment, but the host (development) environment as well. Target systems themselves mainly run the risk of data-related compromise. If the APSE does not operate in a trusted manner, then both data-related and execution-related compromises can occur. Data-related compromises tend, to be relatively non-repetitive. Also, they are subject to control by means of either personnel procedure (ranging up to action as drastic as total physical isolation) or programming rigor (in the sense of trusted computing bases, etc.).

The damage caused by execution-related compromises, by contrast, can be much more far-ranging. Execution-related compromises might, for example, manifest themselves by the inclusion of Trojan horses in the developed code. These would be basically undetectable as an execution problem by the time the target system has been deployed. More subtly designed-in covert channel flows can also be included. Of course, data may be designed into the developed programs themselves (a likely possibility in programs which are very target-specific). In this case, execution-related compromise can eventially manifest itself on the target system as a highly-virulent data-related compromise. <u>Any</u> software development system which is not worthy of trust is a perfect place for execution-related compromises to begin; such a Genie must be regarded as being capable (in the wrong hands) of a great deal of evil.

Suppose that one begins with a safe, secure, trustworthy software development system (e.g. a secure APSE). If one then progresses in a trusted fashion to the target system, then the overall process of project development (considered throughout its life cycle) has a much higher assurance of being secure.

1.4.4 <u>The APSE as a Security Sieve</u>

Currently, no <u>explicit</u> provision for security or trust has been made in the [STONEMAN] specification of the APSE, regardless of the sensitivity of information processed by the APSE. It has been argued that provision may be made by procedural methods, i.e. run the APSE as an entirely closed shop, with every user cleared to see all the information available from the APSE. This corresponds to the notion of "System High" operation (see [5200.28]). In such an event, even if the target system has been designed to be secure (multi-level or otherwise), the untrusted steps in the target software's life cycle still have <u>not</u> disappeared. The development cycle is not clean with respect to security; the dirt is hidden under an APSE.

For any typical DoD organization, the same security considerations exist for program development as exist on the target system. The organization may have a number of development facilities. There may be a need to share these facilities between various projects. Some of these projects may deal with Sensitive Compartmented Information (SCI), and some may not. If they deal with SCI, the compartments will likely be different from project to project.

The user community for these facilities is typically of a broad nature, including both DoD personnel and contractors, potentially not working on-site. As a result, the development facilities may eventually need accreditation for Compartmented Mode and/or operation as a multi-level secure system. To this end, security requirements on the APSE, and on the KAPSE in particular, must be considered. In view of this, there must be the capability for an APSE that, in the rating system of [TCSEC], is worthy of at least a B2 rating. The interpretation of the criteria and requirements in [TCSEC] in the framework of an APSE will be considered in this document as appropriate.

1.4.5 The Secure APSE: Hopes for Controlling the Genie

The APSE design does not lack for concepts of security. The CAIS node-model database is hierarchically organized and makes extensive use of access control lists on a node-by-node level; the discretionary security of the APSE is thus mostly in place already. Much of what will be said here refers to mandatory security considerations (coming up with a scheme of clearances and classifications) and how the existing APSE/KAPSE concepts can be extended to embrace these considerations as well.

1.4.6 A Few Words About Security and Integrity

Given that the APSE is intended to provide for program development and configuration control, it is appropriate to pay attention to integrity requirements. Integrity is in a conceptual sense the dual of security, in that aggregation of information lowers its collective integrity level in the same way that it raises its collective security level.

DoD has no clearly-specified sanctions dealing with integrity violations the way it does for security violations. However, one can do considerable damage (e.g. corrupting or wiping out an entire data base) by ignoring integrity considerations, whether willfully or by simple negligence. Integrity is a natural concern which the APSE concept is intended to address. Security should be just as natural a concern, which the APSE should also address; it is a requirement, but it is also good common sense. This document will address security from both points of view.

1.5 On Pyramids and the Nature of Requirements

Just as security is a relative concept (nothing is absolutely secure, just more secure or less secure by comparison to other entities), so also are requirements, and, in particular, requirements for security, relative concepts. In addressing security requirements on the APSE, this document aims to illustrate the security elements that an APSE needs most to accomodate in order to make the APSE as secure as it has to be.

While pyramids can be built entirely by humans (a documented fact), this technique is excessively labor-intensive (also a documented fact), and most modern construction requirements today would include the provision for machines to aid the process (e.g. cranes and hoists and trucks would be viewed as requirements). Similarly, while an APSE can be made secure by operating it as a dedicated facility and using entirely manual and procedural workarounds to security problems, such a technique is excessively labor- and cost-intensive, and error-prone as well. The security requirements described here are intended to describe what must be done, in the context of security, that still preserves the intent of the Ada initiative.

SECTION 2

BASIC MANDATORY SECURITY REQUIREMENTS

There are certain deceptively simple considerations about security which arise from the paper-handling security world. Their origin is in Executive Order [EO12356], which specifies the need for a system of classification, and DoD Regulation [5200.1R], which provides explicit regulations governing handling of classified information, _irrespective_ of the medium or media in which the information is represented. The notion of mandatory security is intermittently developed _in abstracto_ here, and then given more concrete substance or instantiation in the real/paper and computer worlds.

The primary consideration is that a subject S (be it a person, a process or the more generic "user") can have access to an informational object O only if S is authorized to have access to O and is trusted enough that no action S takes would make O accessible to an unauthorized subject S′, i.e. S will not _compromise_ O.

What happens in the paper world, though, if O loses its value or sensitivity? In this case, there may be a subject S who takes an "authorized" action (in accordance with appropriate policies) so that S′ (or any other subject like S′ who is authorized to know) now does have access to O. S in this case is said to have _downgraded_ O. S is acting as a (_System_) _Security Officer_. S is not considered to have compromised O. The important difference here is one of authorization.

In the paper world, compromises are minimized by such procedures as investigations, non-disclosure oaths and disciplinary actions, as necessary. In a computer world, no clear analogies exist to these procedures. The method for avoiding compromises in such systems is to use highly formal techniques in developing the systems in the first place. This insures that the systems will perform in a trusted (in accordance with policy) and reliable manner.

2.1 Classifications, Compartments and Clearances

2.1.1 _Classifications_

The DoD has a well-defined scheme for dealing with information which is considered sensitive. For the objects (actually, for their real- or paper-world instantiations), there are _classifications_, and for the subjects, there are _clearances_.

For purposes of access restriction, an object is classified in a general way into one of a number of levels of sensitivity. The levels are mandated in [EO12356] and further detailed in [5200.1R]

To apply more stringent restrictions (which afford tighter control against the risk of compromise), an object may additionally be restricted to one or more compartments within a classification. In this case, more compartments mean more restrictions; as common sense might suggest, compartmentation typically occurs more often at higher sensitivity levels.

2.1.2 Clearances

It is important that a person who is granted access to information (of some level of sensitivity) be determined to be worthy of trust that s/he will not compromise that information. A person is cleared of objections, doubts or suspicion regarding his/her trustworthiness. Generally, the granting of this clearance is based on the result of an investigation of some intensity. A person is cleared to have access to informational objects up to a certain classification, and no higher.

2.1.3 Clearances versus the Computational Abstraction

In the computer world, however, a subject's clearance is a different notion. Nothing directly corresponds to a real-world investigation and clearance of a subject. In the small, subjects may be investigated by doing rigorous validation and verification of the code and entities which go into their being, either automatically or manually; this is the raison d'etre behind the technology of trusted computing. However, they are not cleared to a level in the way that people are; they are "trusted", and "trustable" as well, not to compromise (i.e. perform actions contrary to policy) information irrespective of its level.

In the large, systems, as agglomerations of subjects, are subjects themselves. However, accreditation (authority to process) of a system and the granting of a clearance to a person are analogous in only the most basic sense. A system is accredited based on its precise physical and logical configuration and location, on the type of information it is expected to handle, and on the assumption that its "sensitive parts" are operating in a benign environment (an assumption not made of a cleared person).

2.1.4 Classification as Access Set Management

In either a paper- or computer-world framework, clearance has been defined in terms of classification; the real world defines a subject's clearance in terms of the classification of objects to which s/he has access.

For a fixed universe of subjects and objects and a given type of access A, the classification of an object imposes a restriction on the set of subjects who can make access to it. In particular, every object O has assigned to it a "mandatory access set" of subjects, namely the set $M_A(O)$ of subjects whose clearances dominate the classification of O. The classification of O thus manifests itself as a mandatory access restriction which assigns each object a set of subjects. The intersection of this set with the set $D_A(O)$ who are to be granted discretionary access to O yields the set of subjects who are cleared for access to, and have "need to know", O. These are the two conditions recognized in [5200.1] for granting access to classified objects.

In real-world practice, of course, keeping track of the effects of classifications by means of mandatory access lists is much too cumbersome, and at odds with classification assignment methodology. Object marking, in particular, would be completely useless if done by *access set* rather than by level. However, the classification of an object manifests itself naturally in a mandatory access list.

2.1.5 The Classification-Induced Lattice

The access sets described above can be given a mathematical structure. They are subsets of the set A of all people who might conceivably have access to information, and may be partially ordered by the "reverse inclusion" relation. That is to say, an information object O is considered at least as sensitive as an object O' if the access set for O is contained in the access set for O'.

In this way, the access set structure embeds neatly into P(A), the set of all subsets of A, ordered (as above) by reverse inclusion. Suppose we have two elements of P(A), i.e. two subsets (say L_1 and L_2) of A. Then there is a _greatest_ element of P(A) less than or equal to the access set elements L_1 and L_2 (the _greatest_ _lower_ _bound_ of L_1 and L_2), which is the set-theoretic union of L_1 and L_2, Similarly, there is a _least_ element of P(A) greater than or equal to the access list elements L_1 and L_2 (the _least_ _upper_ _bound_ of L_1 and L_2), which is the set-theoretic intersection of L_1 and L_2.

The "top" of this structure is the empty set (corresponding to the practical reality that no one person is authorized to know absolutely everything). The "bottom" of this structure is A (corresponding to the practical reality that there is "information" in the system that is not considered sensitive at all, such as the current date). P(A) is thus a lattice.

If one alters A by the addition or removal of elements (potentially authorized personnel) and gets A′, P(A′) still has a lattice structure in the same way. Alternatively, suppose L_2 is a proper subset of L_1 and both are subsets of A. Then the sets "between" L_2 and L_1 inclusive (normally described as the interval $[L_1,L_2]$) also form a lattice, with "bottom" L_1 and "top" L_2. In fact, the intersection of two such lattices $[L_1,L_2]$ and $[L_3,L_4]$ (the collection of all access sets which are simultaneously in $[L_1,L_2]$ and in $[L_3,L_4]$), it it is non-empty, also forms a lattice. In summary, the lattices described above all fit together in a very natural way, whether by restriction, extension or intersection.

The classification structure can thus be embedded into P(A) and given a lattice structure in a mathematically natural way. Since the determination of object classification is mandated by a security doctrine or policy, controls associated with such access lists are described as mandatory access controls.

2.1.6 Discretionary Access, Teams and Lattices

The APSE has a concept of teams (see [STONEMAN], sections 3.E, 3.J, 4.A.7 and 4.A.9) which is elucidated in its ALS meta-implementation. APSE teams are collections of users grouped together for some purpose, generally related to the process of software development. A given user is identified to ALS by a two-part identification, consisting of a team identification (many team ids are possible, since a user can be on many teams) and a user identification (which is unique for a given user).

Teams are convenient sets of people which are subsets of A. Like DoD classifications, APSE teams can also be given a mathematically natural lattice structure by the same embedding process. In the team case, the intersection of teams would represent the conjunction of project roles.

For example, a given person might be on a software design team and also be on the team dealing with a particular aspect of a project, say, graphic display management. In this case, that person would also be on the "intersection" team, namely the team doing the design of the graphic display manager.

Since the composition of teams (or other access lists) is often more a matter of organizational, need-to-know discretion than of mandated doctrine or policy, controls associated with such access lists are described as discretionary access controls.

2.1.7 Blending Discretionary and Mandatory Access

The important point in the above formal discussion is quite simple. There are two types of access control, namely mandatory (from security policy) and discretionary (from security-oriented or functionally oriented need-to-know). In a formal sense, these two access concepts fit into a common, lattice-oriented framework. In a practical, real-world sense, then, it is good common sense to apply mandatory access controls the same way one applies discretionary access controls.

2.2 What Does This Have To Do With An APSE?

2.2.1 Mandates Regarding Trust, Again

The objects under the purview of an APSE in the current (multi-level classification) context thus require some management scheme. An object must be classifiable, and its classification must be manageable, in a trustworthy fashion. The classification requirement means that there must be some scheme for labeling objects (both internally and externally). The manageability requirement means that there must be some scheme for changing and tracking those objects' classifications and for mediating subjects' access to objects based on classifications and clearances. That is to say: in addition to the requirement for discretionary access control (which is called out in [STONEMAN] 5.A.7), there is also a requirement for mandatory access control in any secure KAPSE.

The access-list view of classifications is not considered here purely as a random mental exercise in introductory computer security technology. It is considered because the APSE data base is implicitly very much organized around the access-list concept; some of the clues concerning how to get from the mandate "the APSE needs to be secure" to the conclusion "the APSE can accomodate security" are contained therein. Additional guidelines for what is needed for an APSE to accomodate security are contained in [TCSEC] and will be considered in the next section.

2.3 Audit Mechanisms

Because security is only relative and not absolute, it must be assumed that in spite of system security precautions, compromises may occur. If the information being processed by the APSE is sensitive, potential compromises must be at least monitored, if not actively suspected. As a result, a requirement exists that any actions taken by any subject in the system which might have an impact on the security of the system must be traceable. Therefore, the APSE must have the ability to audit such actions at a level below that of the tools; the appropriate place for this is the KAPSE.

The audit mechanisms which are called out in [STONEMAN] relate to configuration control and object history, but are not specified in detail. To the extent that security-related actions partially comprise the history of an object, [STONEMAN] does address the questions of audit mechanisms and protection, but it also states in Section 5.B.2 that "the requirement [on the KAPSE] does not dictate the manner in which protection is offered, nor that the protection mechanism must actually be implemented within the KAPSE." Thus, the security audit requirements, while they theoretically blend well with the configuration audit requirements, are not formally spelled out in [STONEMAN]. The degree to which the security audit mechanism integrates with the configuration audit mechanism is partly an integrity issue and partly an implementation issue and as such shall not be considered here. Again, however, as with the basic classification, clearance and management issues described above, guidelines for auditing are contained in [TCSEC] and will be considered in the next section.

SECTION 3

APSE SECURITY REQUIREMENTS

As mentioned previously, the basic security requirements for the APSE are those of a B2 system as defined in [TCSEC]. This section interprets those requirements with respect to an APSE. The B2 requirements deal with mechanisms, assurance, and documentation. Of these, only mechanisms are considered here. Specifically, the B2 security mechanisms which the APSE must accomodate are related to the security policy and accountability requirements called for in sections 3.2.1 and 3.2.2 of [TCSEC], respectively.

In order to interpret these requirements in a concrete fashion, the Softech ALS is used as an example, rather than the relatively less concrete description of an APSE in [STONEMAN]. First, the basic functionality of the ALS KAPSE is described in relation to the other parts of the ALS. Next, an operational view of KAPSE security is given. Finally, specific requirements for discretionary security, object reuse, labels, mandatory security, identification and authentication, and auditing are discussed. Wherever possible, specific reference to requirements in [TCSEC] are made.

3.1 KAPSE Functionality

In order to put the functional purpose of the KAPSE in better perspective, the KAPSE must be recognized to be a part of the APSE. The functionality of the KAPSE is dictated by its purpose as part of the APSE and by the functionality of the APSE. The functionality of the APSE is thus considered first.

3.1.1 How APSE Functionality Determines KAPSE Functionality

The APSE is intended to provide the facilities necessary to bring a software project from inception to completion in a time- and cost-effective way. It is intended to "ease" the problems traditionally associated with a project. These include (but may not be limited to) setting things up in the first place, design, development, component integration, testing, evaluation and configuration management throughout the life of a project. This is done through the use of a standardized central data base, a standardized user-system interface and a standardized set of tools. If the APSE is to be multi-level secure, some tools may have to be trustable in the sense of security. Section 4 of this paper will consider why this is the case and how it may be done.

The primary distinction between one instantiation of an APSE and another is the APSE data base. It contains all the information related to the software, including source, intermediate code, executable code, environment descriptions, documentation, access control, historical journal maintenance/auditing and configuration control. The configuration control part would include maintence of varying revisions of the same software, maintenance of variations of types of software and of dependencies and associations between various objects within the system.

The Ada Language System (ALS) effort initiated by Softech [ALS82] designed an APSE data base which models the data base requirements established in [STONEMAN] and modeled in [CAIS]. It is a hierarchical data base constructed as a directed acyclic graph. This is, in effect, a rooted tree structure which also allows for links across branches as long as they create no direct·d loops. At the nodes are

1. data (which will consist of offspring specifications if the node is a directory node or element specifications if the node is a variation header node),

2. parentage information (both true, created-from nodes and foster, share-oriented nodes),

3. attribute information (a great deal of which is standardized and acts as a workable catch-all for other required information), and

4. associations (a very few of which, in the soon-to-be-described program library, are standardized).

The ALS meta-implementation of the APSE has the standard attributes **node_type**, **category**, **creation_date**, **derivation**, **derivation_count**, **no_access**, **read**, **write**, **append**, **execute**, **via**, **attr_change**, **revision**, **default_variation**, **purpose**, **availability**, **location**, **target**, **acquired_data** and **compatible_targets**. These serve to provide information about node identification, configuration control, targeting, availability (in the sense of being on-line, off-line or archived), node purpose and controlled accessibility.

Program libraries are set up as special nodes. They implement the container concept of Ada which recognizes that packages should be used wherever possible to allow for easy and standard support of higher-level programs. In support of this, the ALS has the standard associations **derived_from**, **depends_on**, **referenced_by** and **acquiring_containers** which define program unit interdependencies.

It is the function of the toolset to provide the higher-level interface between the user of the APSE and the data base itself. We now address the division of labor more closely.

3.1.2 Functional Distinctions Between the KAPSE and the APSE

A distinction is drawn in the APSE between:

1. those tools which are useful for some particular applications or projects (these would include such things as target-specific compilers/linkers, verification tools or text formatters dedicated to project-specific requirements),

2. those tools which are essential for all projects (editors, data base maintainers, and so on) but which are host-independent and required for creation of the tools described above, known collectively as the minimal APSE or MAPSE, and

3. those elements of the APSE which are not necessarily included in the operating system (OS), but which are required to provide the interface between the particular operating system and the user together with the APSE components listed above (those host- or OS-dependent elements are known collectively as the kernel APSE or KAPSE.

The functional and operational considerations for the third of these three classes are now examined.

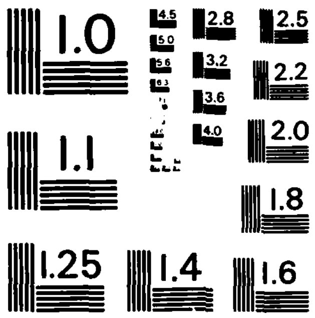
3.1.3 Functionality Relegated to the KAPSE

The KAPSE is responsible, as noted above, for providing the host- or OS-like interfaces to the rest of the APSE. The KAPSE is also responsible for providing the keystroke-level user-system interface and for providing the slightly higher-level command language interface. In particular, the KAPSE is responsible for mediating access to all resources within the APSE.

When a user requests a service, it is the toolset that performs that service. It knows nothing of the implementation details of the APSE in question. To deal with this, the toolset invokes the KAPSE (which may in turn invoke the OS) to perform the basic primitives required, such as changing an attribute, adding a node, altering access, and so on.

AD-A147 648 KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM PUBLIC REPORT VOLUME 4(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 30 APR 84 4/5

UNCLASSIFIED NOSC/TD-552-VOL-4 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

It is the function of the toolset to provide the higher-level interface between the user of the APSE and the data base itself. We now address the division of labor more closely.

3.1.2 Functional Distinctions Between the KAPSE and the APSE

A distinction is drawn in the APSE between:

1. those tools which are useful for some particular applications or projects (these would include such things as target-specific compilers/linkers, verification tools or text formatters dedicated to project-specific requirements),

2. those tools which are essential for all projects (editors, data base maintainers, and so on) but which are host-independent and required for creation of the tools described above, known collectively as the <u>minimal</u> APSE or MAPSE, and

3. those elements of the APSE which are not necessarily included in the operating system (OS), but which are required to provide the interface between the particular operating system and the user together with the APSE components listed above (those host- or OS-dependent elements are known collectively as the <u>kernel</u> APSE or KAPSE.

The functional and operational considerations for the third of these three classes are now examined.

3.1.3 Functionality Relegated to the KAPSE

The KAPSE is responsible, as noted above, for providing the host- or OS-like interfaces to the rest of the APSE. The KAPSE is also responsible for providing the keystroke-level user-system interface and for providing the slightly higher-level command language interface. In particular, the KAPSE is responsible for mediating access to all resources within the APSE.

When a user requests a service, it is the toolset that performs that service. It knows nothing of the implementation details of the APSE in question. To deal with this, the toolset invokes the KAPSE (which may in turn invoke the OS) to perform the basic primitives required, such as changing an attribute, adding a node, altering access, and so on.

It is thus the KAPSE's responsibility to know the "local customs" which are dependent on the host and operating system. In particular, it must know how the local access control is implemented; it has, at a "bit, byte and element" level, the responsibility for providing or denying access to an object in the data base. In summary, if security is to be provided in an APSE, the KAPSE must implement it.

3.2 KAPSE Operational View of Security

We now consider what security an APSE (and a KAPSE, in particular) must provide. The types of security fall into the discretionary and the non-discretionary (mandatory). For the purposes of the current considerations, it is assumed that the access set concept is understood.

3.2.1 User View of Secure APSE / KAPSE Operation

The workings of a secure APSE, as a (human) user might view them, can be modeled in a fairly straightforward manner. The brief description which follows presents a usage model somewhat after the fashion of Landwehr and Heitmeyer in their description of a security model for military message systems in [LAND]. Terms which have special meaning in the context of this document are printed in upper case.

3.2.1.1 Establishing Access to the APSE. A human USER can gain access to the APSE only by logging in. To log in, a USER presents a USERID (user identification) and the system performs authentication, using appropriate techniques.

3.2.1.2 Usage of the APSE. Following a successful authentication, the USER may invoke TOOLS (or PROCESSES) to perform the general functions provided in the APSE. By invoking TOOLS, the USER may manipulate (viewing, modifying, referencing or otherwise making access) OBJECTS at NODES. A TOOL may, in turn, invoke another TOOL on the USER's behalf, to accomplish the desired objective.

3.2.1.3 Access to TOOLS. TOOLS (in the CAIS model) are distinguished OBJECTS. A TOOL is at a NODE which specifies that the OBJECT is to be invoked (executed) as a TOOL. The TOOLS a USER is authorized to invoke depend on

1. The USERID of the USER and

2. the access and security_level ATTRIBUTES (which must show that the USER is authorized for access) of the NODES

corresponding to those TOOLS.

3.2.1.4 Access to OBJECTS. The OBJECTS a USER is authorized to manipulate depend on

1. the USERID of the user,

2. the security_level and working_range ATTRIBUTES of the NODES corresponding to the USER,

3. the access, security_level and security_range ATTRIBUTES of the NODES corresponding to the OBJECTS, and

4. the TOOL which is invoked to make access to the OBJECTS.

The APSE as a whole (and the KAPSE in particular) must enforce the security assertions implicit in the security requirements listed in the rest of this section and in the next section. That is, the KAPSE prevents the USER from invoking TOOLS in a way that would contradict these assertions.

3.2.2 Discretionary Considerations

Discretionary security is implemented via access lists. They control which subjects (be they users, tools or other kinds of subjects) are authorized to read, write, execute, append, or otherwise make access to a given object.

It must be remembered that a (human) user's interface to an APSE during a session is via the terminal. Terminals may not be "trusted" because of their location. This necessitates a concept of permitted working access, which depends upon both the user and the terminal.

A user may be on many APSE teams at once. During the course of an APSE terminal session, the user will have a "working team" (much like a working directory). The user may change the working team as long as the requested new team contains that user as a member. The user has a session-independent working range of access lists, namely the set of all access lists of which s/he is a member.

It may, however, be that there is information that is so discretionarily sensitive that, for a particular terminal and session, that access cannot be granted to it, even for a user on the APSE access list. For instance, if several contractors have proprietary software on a (possibly distributed) APSE, it may be undesirable for even a DoD project manager to be able to examine contractor X's software at contractor Y's site, regardless of the project manager's oversight responsibilities and consequent APSE-encoded privileges. The terminal itself thus has a working range of discretionary access, and the user's effective working range during a particular session must be restricted by the terminal's working range. Discretionary access is granted only if both the user and the terminal which mediates his/her access have the necessary discretionary access privileges.

Discretionary access privileges must be set up, set, reviewed and reset, sometimes in ways the creator of a given object has no privilege to effect. This may happen because the creator of the object is no longer connected with the project, or has changed responsibilities and is no longer responsible for the disposition of the object in question. In such events, a system administrator is responsible for these tasks in an ALS. Such a privileged user must exist to manage discretionary security.

It is the KAPSE which is responsible for validating, permitting and effecting the requested type of access to an object. The KAPSE must have cognizance of ALL the entities in the information flow path. This includes the terminal. Hence, it is the KAPSE that must know, adjust and report (if necessary) the working level, in addition to checking for legal access.

3.2.3 Mandatory Considerations

Mandatory security shares many attributes with discretionary security when considered from the point of view of access lists. The general character of the considerations for mandatory security in the KAPSE is much the same as that for discretionary security. Rules for mandatory security are set by policy (which is independent of the individual object under consideration) rather than by need-to-know considerations.

For a fixed subject, operating through a fixed, specific chain of tools, the validity of an access is dependent upon the type of access and the classification of the object, rather than upon the type of access and the identity of the object. That is, for a given (fixed) type of access, the "access potential" will be the same for any two objects at precisely the same classification level. This distinguishes mandatory security from discretionary security. The only way to alter this from access attempt to access attempt is to change the classification of the object. Only a certain type of subject, namely a System Security Officer, (SSO) is authorized and trusted to do this arbitrarily. In a secure system, it may in fact be appropriate for the SSO role to be filled by the System Administrator as well. Conceptually, SSOs are required in the APSE to make mandatory security workable.

There must be a working level of discretionary access for a user at a terminal which answers the question "what does this user have access to now?". There must also be a working level of security or clearance for that user at that terminal. Again, this level may change within constraints applied (at a minimum) both on the terminal and on the user. The rules for working levels are described later in this paper.

The extent (interval) in the lattice over which the system permits the working level of an APSE entity (whether user, tool, or object) to range will be called the _working range_. The concept of a working range in an APSE is a useful one. Many APSE tools may have to make access to information widely scattered through the APSE data base. That information may be very closely associated with information of a different classification. Unnecessary downgrading will be avoided if some mechanism exists to respect the different classifications of these associated pieces of information. The application of the working range concept to address the unnecessary downgrading problem will be discussed at somewhat greater length in the next section.

Except for the working team, it is generally unnecessary to describe to the APSE user his/her current discretionary access level. The contrary is true about mandatory access (security) level. Because it is mandated by policy, the working level is always well-defined. It is interpreted by human users who must act accordingly when dealing with classified data. Knowledge of the working level is necessary; because the level is simply described (in comparison to discretionary access level), it is practical to present.

When dealing with untrusted tools, a user should be able to reliably determine the level at which s/he is working. Since the KAPSE provides the user the interface to the rest of the system, it is the responsibility of the KAPSE to provide this information to the user via a "trusted path" that can be invoked by the user in a known way.

The same argument which concludes that the KAPSE must provide the discretionary security for an APSE also concludes that the KAPSE must provide the mandatory security for an APSE. It should also be stated, however, that access to an object by a subject can only be granted if _both_ discretionary _and_ mandatory access constraints are satisfied.

As was noted at the end of Section 2, the "relativity" principle of security (that is, compromises are theoretically possible) mandates the auditing of events or actions in the APSE which might have an impact on the security of the APSE. Essentially, _anything_ which has an impact on the life cycle of an object (creation, destruction, reading, writing, appending, attribute and association modification) might have impact on the security of the APSE. Hence any such events or actions must be audited; what was done, who did it and when are the critical details of these events or actions. This requirement is called out in section 3.2.2.2 of [TCSEC].

3.3 KAPSE Requirements From (TCSEC)

The B2 security requirements for a KAPSE as called out in [TCSEC] are now considered, on a point-by-point basis.

3.3.1 KAPSE Discretionary Access Control ([TCSEC] 3.2.1.1)

For a B2 system, [TCSEC] calls for access controls which are capable of including and excluding access to an an object to the granularity of a single user. As noted in a previous section, the APSE does have a concept of access control mandated in [STONEMAN], and the ALS meta-implementation of this is the set of access list attributes associated with every node. Thus, the B2 requirements for discretionary access controls are met by the ALS without modification. Access lists are directed toward meeting the needs for access control (albeit discretionary in the ALS meta-implementation) at the object level and for project team recognition. A typical access list in an ALS might in fact be a team, and every user is on some collection of teams, to one of which s/he defaults when initiating a session.

In the ALS, there are seven (7) access lists associated with each node/object. These are

1.	no_access:	anyone on this list is automatically denied access of any sort to the object in question.
2.	read:	anyone on this list can read from the object in question.
3.	write:	anyone on this list can write to the object in question.
4.	append:	anyone on this list can write to the end of the object in question.
5.	attr_change:	anyone on this list can change the node/object attributes (subject to additional controls).
6.	execute:	anyone on this list can execute the object.

7. via: any <u>tool</u> (which, recall, can act as a subject) on this list has access to the object. This is for special nodes.

The access controls that are described here are also geared to proper maintenance of the APSE as a programming system in the more "traditional" sense. The concept of Program Libraries (PLs) must be well-supported by the system. An arbitrary tool may not be aware of the PL structural considerations inherent in the program language constructs which Ada has. Carefully mediated access to the nodes in the APSE data base must be maintained so that an "unknowledgable" user or tool does not destroy the integrity of the PL.

3.3.2 <u>KAPSE Object Reuse ([TCSEC] 3.2.1.2)</u>

When created, KAPSE objects must contain no data for which the creating subject is not authorized.

3.3.3 <u>KAPSE Labels ([TCSEC] 3.2.1.3)</u>

Mandatory sensitivity labels must be associated with all APSE subjects and objects, including:

a. Users of the system;

b. Program library elements in source form;

c. Program library elements in compiled form;

d. Text files, as for documentation;

e. Directory nodes (whose entries might be classified); and

f. any other data associated with a node in the data base hierarchy.

These labels are to be used as the basis for mandatory access control decisions, and must accurately represent the security level of the entity with which they are associated. The labels must be exportable outside the realm of the relevant APSE, regardless of the receiving subject. The labeling requirement applies to electronic exports from the APSE to other electronic subjects (typically another computer); it also applies to exports to a human user (i.e. what the user sees at a terminal), and to hard-copy "instantiations" (listings or other graphic output) of objects from the APSE. In particular,

a. machine-readable labels must be placed on tapes, disks, and computer-to-computer communications links,

b. the maximum level of a user's session, and hence his/her terminal, must be shown to the user upon login, and must be available to the user <u>whenever</u> it is requested;

c. there must be labels at the beginning and end (top and bottom) of each page indicating the highest level of classification or sensitivity of the information on the page, and

d. there must be labels at the beginning and end (first and last page) of each listing indicating the highest level of classification or sensitivity of the information in the listing.

The SSO must be able to (in an audited fashion -- see "Auditing in the KAPSE") change or override the labels which are so associated with all the informational objects mentioned above. This includes the hard-copy labels.

3.3.4 <u>KAPSE Mandatory Access Control ([TCSEC] 3.2.1.4)</u>

The KAPSE must enforce mandatory access control policy over all resources that are directly or indirectly accessible by subjects external to the KAPSE. It must support both hierarchical (level) and non-hierarchical (as in compartments) classifications, and should in fact be able to support at least two levels of security.

As suggested earlier in this document, it is appropriate to apply the mandatory access controls in the data base at the same place as the discretionary access controls. That is to say, an appropriate place to apply security classification of objects is at the node level, by use of a new <u>mandatory</u> attribute called **security_level**.

The precise meaning and format of the entries in the security_level list must be universally agreed upon if transportability of portions of an APSE data base is to be possible in a secure manner. One workable scheme is to make the first element of the list the level (unclassified, confidential, secret or top_secret) and make the rest of the entries the compartments, coded in a standardized fashion. This information

1. should be settable by the creator (and <u>only</u> if the level set is within the creator's working range),

2. should default to the level of the parent node (again, only if the default is within the working range), and

3. should not be downgradable below its created level except by a system security officer.

3.3.5 KAPSE Identification and Authentication ([TCSEC] 3.2.2.1)

The KAPSE must require users of the APSE to identify themselves before any actions are performed on their behalf. Some form of trusted authentication (in a trusted fashion, referencing secure authentication data) must be provided by the KAPSE. The identification of a user which is so provided must be associable to the auditable (see below) actions taken by that user. What this means is that a trusted path to the trusted computing base (within or underlying the APSE) must be available to the user in a known fashion. In particular, this path must be available at the time of establishment of the initial connection, i.e. logins must be handled by the KAPSE in a trusted fashion.

3.3.6 KAPSE Auditing ([TCSEC] 3.2.2.2)

The KAPSE must be able to create, maintain and protect a record of access to APSE objects. The record must identify

- the object in question,
- what subject requested (attempted or obtained) access to the object,
- what type of access was requested, and
- when the access request occurred.
- The System Administrator must be able to selectively audit access either subject-wise (actions of one or more subjects based on identity) or sensitivity-wide (accesses to objects of a certain security level).

Also, the KAPSE must be able to audit all events that might contribute to exploitation of covert storage channels. That is to say, if memory is being (even logically) shared by two processes in such a way as might disclose information in an unauthorized manner (e.g. by inter-process communication), then that event must be audited by the KAPSE.

SECTION 4

TRUSTED TOOL AND WORKING RANGE CONSIDERATIONS

This section provides some justification for requiring trusted tools (operating outside the KAPSE), working levels and ranges, security levels and ranges, some examination (through some simple generic examples or scenarios) of the interaction of these concepts, and rules, founded upon the basic mandatory requirements given in Section 2, governing the operation of levels and ranges. Generic examples are provided first.

4.1 Some Examples

Suppose that a project effort has an APSE, possessing several levels of data. Suppose further that the following conditions hold.

1. The APSE contains information at the unclassified, confidential and secret levels.

2. The APSE contains planning documentation information, a large number of separate, project-unique program units in Ada source code and in (perhaps by virtue of compilation) intermediate language pseudo-code.

3. Each of the entities above has information at each level, e.g. there are unclassified, confidential and secret informational objects (in the abstract sense) in the planning documentation information,

4. The classification of type information for confidential and secret Ada routines is typically of strictly lower classification than that of the code. For instance, if there is a procedure INTERROGATE_EMITCHAR that is classified secret as a whole, the specification of types of its arguments may be classified only confidential.

5. The classifications of the paragraphs in the planning documentation may vary rapidly from unclassified all the way up to secret.

We now consider two simple development tasks for a user with enough expertise to perform both tasks, but only a confidential clearance:

1. excerpting unclassified documentation, from planning information whose highest classification is secret, to aid in production of more detailed specifications, and

2. running a consistency check (a partial compilation, without linking, to make sure invocations of other program units are properly typed) on an unclassified monitor program unit that calls confidential and secret program units whose type information is never above confidential level.

If one ignores the aggregation issue (as it has been ignored throughout this paper), it is clear that the first task involves the user examining that portion of the data which is classified no higher than confidential (since the user is only cleared to that level) and producing excerpts which are unclassified. Likewise, the second task involves referencing confidential or unclassified information which may be closely associated with, but which does not disclose, information of a possibly higher classification.

In both cases, the user is attempting access only to information for which s/he is authorized, and so should not be impeded by the system from task performance for security reasons. However, in both cases, a simple organization of data that does not separate the various levels of information classification (albeit in a trusted fashion) will prohibit the user's accomplishment of the task.

Although these tasks can be accomplished by having them "forwarded" to a human agent with higher clearance and subsequently having the output downgraded back to the user, this process will take too long. If, however, there is a <u>trusted</u> tool providing the interface to the multilevel objects, that tool can examine the working ranges and working levels of its requestors (via a standardized invocation of the KAPSE) and, because it is trusted, make the appropriate accesses to the multi-level files requested. The exact structure of the multi-level object is irrelevant (in that the user, or any of the untrusted tools invoked by the user need not know the structure), <u>provided</u> that <u>direct</u> access to multi-level data objects is suitably limited to the appropriate trusted tools. This can be accomplished by using something like the ALS via attribute. General (i.e. by other than via trusted tools) read access is not really a threat to security or integrity, since the object would have the security_level attribute set at the level of the highest classification of any information contained in it. However, general write access would pose an integrity difficulty, since an arbitrary tool could not be trusted to preserve the multilevel structure.

Structuring of multilevel objects is neither mandated nor specified here. However, the working level and range of a tool and its of the highest-classified information object within the data object (i.e. highsl is the value of the object's security_level attribute).

4.2.2.3 Directory Security Range. For a directory structure node, the security_range attribute will have two entries like the one in the security_level attribute, indicating the lower and upper limits on the classification of data below that node.

4.2.2.4 Default Node Security Range at Creation. The security range of a directory node defaults to that of its parent node and must otherwise be a restriction of the security range of the parent node.

4.2.2.5 Changing Directory Security Ranges. Changing the security range of a directory node is a restricted action, as follows:

a. It cannot be restricted to a range more narrow than that of its existing children.

b. It cannot be changed by any user other than a System Security Officer or the creator of the node.

c. Expansion of the security range past the security range of the parent cannot be done by any user other than a System Security Officer (since this forces parent node ranges to be expanded).

d. Any such changes must be audited.

4.2.2.6 User Perception of Security Range. For a multilevel data object, its security range is of "visible" use only

- to trusted tools, and

- for informational purposes to authorized users,

since they are the only entities in the system trusted to deal with multilevel objects securely. It is intended only for such multilevel objects as documents or program libraries, and assumes trusted maintenance of partitions between subobjects (paragraphs, procedure argument type lists, etc.) of different classifications.

4.2.3 Rules for the Working Range Attribute

4.2.3.1 Working Ranges for Potential Subjects. Users, terminals and tools have reading and writing working ranges. The reading working range is the range of levels of objects from which information may be acquired by the user, terminal or tool. The writing working range is the range of levels of objects to which information may be transferred by the user, terminal or tool. Inherent working ranges are ranges which exist independent of APSE session-related restrictions or alterations. Ranges which are dependent upon APSE session-related restrictions or alterations are described as session or actual working ranges, as appropriate.

4.2.3.2 User Reading Working Range. The inherent reading working range of a user is [unclassified, usrcl], where usrcl is the clearance of the user. A user may restrict or expand the session reading working range during a session, as long as that session working range is within the user's inherent reading working range.

4.2.3.3 User Writing Working Range. The inherent writing working range of a user is [unclassified, syshi], where syshi is the maximum information classification level in the APSE. A user may restrict or expand the session writing working range during a session, as long as that is within the user's inherent writing working range.

4.2.3.4 Terminal Reading Working Range. The inherent reading working range of a terminal is [unclassified, mtwl], where mtwl is the maximum terminal working level.

4.2.3.5 Terminal Writing Working Range. The inherent writing working range of a terminal is [wl, syshi], where wl is the working level and syshi is the maximum information classification level in the APSE.

4.2.3.6 Trusted Tool Working Range. The inherent working range of a trusted tool (for both reading and writing) is [unclassified, syshi].

4.2.3.7 Untrusted Tool Working Range. The inherent reading working range of an untrusted tool is [unclassified, wl]. The inherent writing working range of an untrusted tool is [wl, syshi]. The inherent working range of an untrusted tool is the (almost) trivial range [wl, wl].

4.2.3.8 Restrictions of Working Range Between Subject Entities. The actual working range of a user U during the invocation of a tool T is dependent upon

a. the user U's session reading working range $R_r(U)$ and session writing working range $R_w(U)$,

b. the terminal's inherent working range, and

c. the working range of T and all the tools invoked directly or indirectly by T.

What should be made clear here is that a terminal is not the only entity that mediates a user's access to information. Any tool invoked either directly or indirectly by a user also mediates access to information. The illustration entitled "Subject/Object Information Flow from Tool to Tool" shows a typical high-level flow path for this information. It begins with the user and continues through the user's terminal, to Tool 1, then Tool 2, then Tool 3 (which could be the same as Tool 1) and finally back through the terminal to the user. No data objects are shown in the diagram, but access to them is controlled as described below. In this diagram, rwr stands for reading working range, and wwr stands for writing working range, for the entity in question.

For information to pass from one tool to the next, the intersection of the actual reading working range of the subject tool with the actual writing working range of the object tool (symbolized by the star in the flow path) is taken. Only information whose level is within the resulting intersection can pass. Note that for untrusted tools, this intersection consists in the working level wl only. The overall effect is that the actual working range is restricted by the working ranges of all the entities through which information must pass, and that if any intersection is ever null (i.e. no levels exist in the working range), the inherent invocation must fail. The rules for tool access to data objects are discussed below.

4.2.3.9 Limitation of Working Level. At all times, the working level must be within the actual working range.

4.2.4 Additional Rules Pertaining to Multilevel Objects

In documents, the paragraph markings may vary over a wide range. In program libraries, the existence and types of objects and/or subobjects may be of a different level than that of the object itself. For reference and type resolution, information of a lower classification would be needed than would be needed for full linking. It is envisioned that any secure APSE implementation might have multilevel PL containers maintained in some trusted fashion, to allow for secure, yet efficient, software development.

However, U's actual session working range restricts a trusted tool T's perceived working range when acting on U's behalf. In particular, for read access, the actual reading working range for the trusted tool operating on a multilevel object M will be the intersection of M's security range with the U's actual session reading working range. Similarly, for write access, the actual writing working range for the trusted tool operating on a multilevel object M will be the intersection of M's security range with the U's actual session writing working range. For either reading or

```
                                  |
  terminal -----[wwr(t)]---*---[rwr(tool1)]-----> tool1
                                  |
      ^                                             |
      |                                             |
      |                                       [wwr(tool1)]
[rwr(terminal)]                                     |
      |                                            -*-
     -*-                                            |
      |                                       [rwr(tool2)]
  [wwr(user)]                                       |
      |                                             |
      |                                             v

     user                                         tool2

      ^                                             |
      |                                             |
      |                                       [wwr(tool2)]
  [rwr(user)]                                       |
      |                                            -*-
     -*-                                            |
      |                                       [rwr(tool3)]
[wwr(terminal)]                                     |
      |                                             |
      |                                             v
                                  |
 terminal <-----[rwr(t)]---*---[wwr(tool3)]------ tool3
                                  |
```

Figure 1. Subject/Object Information Flow from Tool to Tool

writing, if the resulting intersection is null (i.e. no levels are in both M's working range and U's actual reading or writing working range), then that type of access to M by T must be denied, even though the tool in question is trusted.

The working range of M is of little significance to an untrusted tool D. D can only be trusted to perceive and act upon M as a single-level object at the highest level of classification of any subobject in it. Thus, the actual reading and writing working range of M when access to it is attempted by D will just be [highsl, highsl].

SECTION 5

SECURITY AND THE COMMON APSE INTERFACE SET

The Common APSE Interface Set (CAIS) is a set of Ada packages "designed to promote the source-level portability of Ada programs, particularly Ada software development *tools*." [CAIS] *This section* provides an introduction to the CAIS in sufficient detail to then consider the security ramifications of the CAIS itself.

5.1 Purpose and Scope of the CAIS

In theory, the CAIS provides all the interfaces to traditional operating system services that are required. If a tool is written in Ada, using the interfaces provided in the CAIS, it should be possible to port that tool from one host system to another and have it be operational without any significant modifications. The CAIS is undergoing constant development, review and revision. Thus, the details of how the CAIS accomplishes its purpose are in a state of flux as of this writing.

5.1.1 The CAIS Node Model

As mentioned earlier in this document, the CAIS is, at its heart, based on a node model. The CAIS specifies all entities defined by.the CAIS which are not explicitly defined in the Ada language itself can be viewed in terms of nodes. Packages are defined in the CAIS to provide all the necessary management of the nodes upon which the CAIS is founded. The CAIS specifies four kinds of nodes.

5.1.1.1 Structural Nodes. These nodes are without content. They serve to "glue" the rest of the nodes in the CAIS (and the system underlying the CAIS) together.

5.1.1.2 File Nodes. These nodes serve to implement files. In them are both files and information about those files.

5.1.1.3 Device Nodes. These nodes serve to provide basic device input, output and control which is not file-oriented (e.g. terminals).

5.1.1.4 Process Nodes. These nodes serve to implement program (and thus tool) invocation, job control and process families. They are created as processes are invoked, hold executable instances of Ada programs (including all their tasks) invoked for the process, and are destroyed after those processes terminate (for whatever reason).

5.1.2 Relationships, Relations, Contents and Attributes

Since nodes are, in themselves, quite abstract and disjoint, additional constructs are needed in order to make nodes usable. The constructs provided in the CAIS are relationships, relations, contents and attributes.

5.1.2.1 Relations and Relationships. Relations and relationships provide the logical connections between the nodes. A relation is a particular directed graph on the nodes of the CAIS. It is uniquely identified by a relation name, which is an Ada identifier. Because of this, relations and their names are frequently referred to interchangeably. A directed edge in the graph of the relation is a relationship. In that a directed edge can be described by the ordered pair of nodes which it connects, a relationship can also be described in terms of those nodes. In this context, the first node is called the source node and the second is called the target node.

For example, one node being the parent of another constitutes a relationship between those two nodes. Thus, the node "family tree" with the source nodes being the children and the target nodes being the parents is the graph corresponding to the "parent" relation. This relation is functional, i.e. a given source node is connected by that relationship to only one target node. In this case, a child has one parent, so that the concept of the parent of a node is well-defined.

However, a parent may have many children. If one takes the family tree example above and reverses all the edges, one gets the graph corresponding to the "child" relation. In this relation, a given node (parent) may have a relationship with many nodes (children). In this case, the relationship will not be functional in nature: "the child" of the given node will not be a well-defined concept. To distinguish among the many nodes having a relationship with the given node, another concept is required, namely the relationship key. The relationship key is an identifier which may be null if the relationship is functional (as in the example of the parent relation). Like the relation name, the relationship key is an Ada identifier.

The parent and child relationships described above are examples of primary relationships. A primary relationship is one for which the graph of the corresponding overall relation is a strict tree. For a primary relation (in which the relationships are all primary), at most one directed path, representing a series of relationships, will exist from one node to another. When a node is created, a primary relationship from its parent node is also established. The parent relationship is thus referred to as "the primary relationship".

All relationships which are not primary are called secondary. Secondary relationships may express connections of an arbitrary sort, such as connecting source code to documentation, alias naming, or tying of user jobs to user directories. If a node is deleted, its primary relationship (with its parent) is broken, and access attempts via other relationships (secondary or otherwise) will fail.

There are three pre-defined secondary relations which associate to each process node. They are:

1. CURRENT_JOB This node points to the top-level job node for the job which created the process.

2. CURRENT_USER This node points to the current user's top-level node.

3. CURRENT_NODE This node points to the current node of focus for the process.

A concatenation of relation names and/or keys emanating from one node to another via a relationship path forms a pathname. Pathnames provide the syntactic navigational means for a process (ultimately acting on a user's behalf) to make access to nodes. One may use the CAIS-defined secondary relations to do this or use a relation defined elsewhere (perhaps in some package separate from those packages in the CAIS).

5.1.2.2 Content. Content is associated with all but structural nodes and is semantically dependent on the kind of node involved. For file nodes, it is the representation of an Ada external file. For device nodes, it is the representation of some logical or physical device. For process nodes, it is the representation of the execution of a program.

5.1.2.3 Attributes. An attribute is associated with one node or relationship, in a functional fashion, and expresses some quality of the node or relationship. The CAIS currently defines only the access_control and security_level attributes, and only defines these on nodes (ALS has a somewhat move extensive suite of node attributes, as described earlier).

5.1.3 CAIS Process Command, Control and Communication

The CAIS provides means for managing process nodes. A process may create other processes, control those processes (suspending, resuming, interrupting, aborting or terminating them) and communicate with those processes.

5.1.3.1 Process Creation. The CAIS provides means for a process to invoke another process and wait for it to complete, or to spawn another process and continue without waiting for it to complete. Provision is made in either case to pass parameters from the creating (parent) process to the created (child) process, adjust the child process's notion of the current node and specify standard input, output and error files.

If the child process is invoked, the parent process is suspended until the child process terminates or aborts, and both results and a completion status may be returned by the child process. If the child process is spawned, the parent process and the child process run in parallel, using CAIS process communication and interruption packages to interact. An AWAIT_PROCESS capability exists which allows the parent process to wait (for some time duration) on the child process as if the parent process had invoked the child process rather than spawning it.

5.1.3.2 Other Process Control. A CAIS process may suspend, resume, or abort another process, or terminate itself in the "normal" manner. If it aborts another process, the aborted process's descendants are also aborted recursively. In addition, one process can interrupt (signal asynchronously in the manner of a pseudo-interrupt) another process. In this case, the receiving process may ignore the interrupt, abort execution, awaken a suspended task or place the interrupt on hold. Provisions exist in the CAIS for determining the status or results of all such actions as appropriate.

5.1.3.3 Process Communication. A CAIS process may communicate with another process (or itself) using channels named with an Ada identifier.

5.2 Security Considerations in the CAIS

Some security ramifications of the CAIS are now considered.

5.2.1 Signaling Between Processes

In the absence of any other conditions of process trust, signaling between processes, as defined in section 6.5.1 of [CAIS], must be confined to the same security level. Unanswered "signaling up" (signals with no response) from a process node at a lower level to a process node at a higher level in the classification lattice presents no security problems. However, "signaling down" (to a process node not at the same or higher level, regardless of whether a response is given), cannot be done by other than a trusted process node.

Process status may in some circumstances serve as a slow, self-clocked, covert timing channel. However, randomization of response time to a process status request can be used to slow down the rate of leakage. In any case, the leakage rate is likely to depend heavily upon the precise CAIS/APSE application and implementation, and ties closely with performance.

The question here is whether a distinction should be drawn between the security level (or working range) of a process node and the level/range of the node's status.

Suppose first that one does draw a distinction between the level of a process node and the level of its status. Suppose further that process node A has access to the status of process node B, and that process node C, operating at a higher security level than A, has control of the status of B, using the CAIS_PROCESS_CONTROL package in CAIS Section 6.2. Then C can use the status of B (say, using SUSPEND_PROCESS and RESUME_PROCESS as 0- and 1-bits respectively) to as a covert channel to A. Even if B is "trusted", B cannot affect this situation. The issue then becomes one of permissible covert channel leakage rate and means of leakage detection (which should be possible, although not automatic, in a B2 system).

Now suppose that one does not draw a distinction between the level of a node and the level of its status. Then one has presumably "solved" the problem of process status as a covert channel, but one may have affected interoperability if (in the context of the above example) an application requires that A be capable of reliably determining the status of B.

In the context of the working range, it is conceivable that process control might be regarded as data from the controlling process node to the controlled process node (even through the controlled node might not perceive it as such). In this case, unless A were trusted (in which case the covert channel problem presumably does not exist), the status would fail to get through to A, since the status of B is actually a "datum" from C. Status information would have to be explicitly recognized BENEATH the CAIS (at the KAPSE level), however, so that C could not "sneak one by" through B.

The response from a higher level to a signal at a lower level poses the same tradeoff between covert channel acceptance and interoperability considerations. One solution might be a toolsmith's guideline saying that a tool intended to work without modification in an Mult-Level Secure system should be ready to handle a security exception (or some other kind of access exception, if the raising of a security exception would in itself provide a covert channel.

5.2.2 Data Sharing

It is possible that a significant amount of data must be shared between processes. This may happen, for instance, in the passage of "sanitized" text or package type declarations in the example given in the previous section. If a reasonable level of performance is required, a mechanism should exist that allows for efficient passage of data as above

- from a higher-level trusted process node to a lower-level (not necessarily trusted) process node, or

- from a lower-level process node to a higher-level process node, with neither process necessarily being trusted.

However, in any secure implementation, such use of information passing should be considered distinct from brief status signaling between processes (slightly longer than that described in [CAIS] Section 6.5), although at least the same security restrictions probably ought to hold for data sharing transfer. It should be noted that, as currently proposed, the CAIS_PROCESS_* packages make no distinction between large and small messages (since even those notions are highly application-sensitive).

In practice, of course, compromise of wide-bandwidth data is significantly more harmful than it is for narrow-bandwidth data. As with signaling, above, this is a leakage-rate issue which ties closely to performance considerations.

5.2.3 Process Isolation

In general, process nodes at different security levels must be isolated from one another, irrespective of trustedness, since untrusted "code", by definition, cannot be trusted not to "walk on" or otherwise subvert trusted code. The data sharing described previously constitutes a limited exception to this policy, as does the status interrogation.

In a typical secure system, though, there is no reason why one process should know anything about another process which it has no authorization (as determined by access control) to know. Hence, no other exceptions to such a policy are sensible.

5.2.4 Attributes and Relationships

In some sufficiently classified DoD programs, existence of certain components of the program, or of the program as a whole, may be classified. Furthermore, certain associations between various aspects of the program may also be highly classified.

The ability to determine the associations described above is quite useful for cleared users with the necessary need to know, but should be denied to all others (to the point of restricting path navigation). Thus, the CAIS should accommodate the ability to apply the security_level and access_control attributes to relationships, as well as to structural nodes.

5.2.5 On Reality and Performance ``Requirements''

Process family management, which is a convenient way to build component tools in a working environment such as UNIX, an APSE or even (perhaps) an Ada run-time environment, is nonetheless extremely expensive in the current technology of secure systems.

Much of the overhead in keeping a secure system secure relates to the need for separation of process contexts in that system. Process creation is expensive because

(a) Context creation is expensive. The control structures which must be set up to provide for secure execution of a process and which define the process context are typically extensive.

(b) "executable instantiation" of a process is expensive. A process must be created in a secure fashion, recognizing the existing process world and its access restrictions.

For the same reasons that context creation tends to be expensive, context switching also tends to be expensive. This can be avoided to an extent if there is the sort of early-in-design optimization that, for instance, the Nebula-style Military Computer Family architecture is intended to provide.

Finally, process management, apart from invoking, spawning, terminating or aborting, is expensive because

(a) context switching is expensive, and

(b) process swapping (which will be a reality in any memory-intensive situation where "enough" memory is not cost-effective) causes substantial overhead in a secure system and is therefore expensive.

Thus, it must be realized that the performance issues which underlie any secure system will have an adverse effect on the implementation of any model similar to a process node model.

It is not, for the purposes of the CAIS, a reasonable conclusion that "we should then throw out the process model". A more constructive approach would be that "considerable attention should thus be devoted to how to manage processes securely AND efficiently".

It may be noted that the handling of processes is not an issue involving requirements and criteria as much as it is a performance or guideline issue. However, this issue should be addressed promptly to avoid doomed efforts to produce cost- and performance-efficient implementations of products in Ada, using a Multi-Level Secure APSE.

REFERENCES

ADAINIT	Department of Defense, Instruction 5000.31, November 1976.
5200.28	Department of Defense Directive, "Security Requirements for Automatic Data Processing (ADP) Systems", DODD 5200.28, revision 2, April 1978.
5200.1R	Department of Defense Regulation, "Information Security Program Regulation", DODR 5200.1-R, August 1982.
STONEMAN	Department of Defense, "Requirements for Ada Programming Support Environments: STONEMAN", February, 1980.
SOWALSN	Department of the Navy, "Statement of Work (SOW) for Ada Language System / Navy", N00024-83-PR-21808, August 1983.
SSPECALSN	Computer Architecture Branch, "System Specification for the Ada Language System / Navy", Naval Ocean Systems Center, San Diego, CA, June 1983.
CAIS	KIT/KITIA CAIS Working Group, "Draft Specification of the Common APSE Interface Set (CAIS)", Version 1.1, September, 1983.
TCSEC	Department of Defense Computer Security Center, "Department of Defense Trusted Computer System Evaluation Criteria", CSC-STD-001-83, Fort George Meade, MD, August 1983.
BELLAP	D. E. Bell and L. J. LaPadula, "Secure Computer Systems", ESD-TR-73-278, Volumes I-III, AD A770768, AD A771543, AD A780528, The MITRE Corporation, Bedford, MA, November 1973 - June 1974.
EO12356	Executive Order 12356, "National Security Information", April 1982.
ALS82	Communications Electronics Command, "System Specification for the Ada Language System", U. S. Army, August 1982.
LAND	C. E. Landwehr, C. L. Heitmeyer, and J. McLean, "A Security Model for Military Message Systems", in preparation.

REFERENCES

(Concluded)

DION81 L. C. Dion, "A Complete Protection Model", IEEE 1981 Symposium on Security and Privacy, Oakland, California, April, 1981.

DISCRETIONARY SECURITY MECHANISM FOR THE CAIS

KIT/KITIA CAIS WORKING GROUP

Abstract

Discretionary security protection and a mechanism for discretionary access controls for the Common APSE Interface Set [CAIS] is described.

Section 1
Introduction

In a trusted computer system, access control mechanisms provide control over access by an individual, or a process acting on the behalf of an individual, known as the subject of the access, to a unit of information, known as the object of the access. Discretionary security protection provides protection from unauthorized access of objects by named individuals or groups of individuals. The establishment of access rights to an object is performed by an authorized individual, typically the creator or owner of the object.

Discretionary control differs from mandatory control because it "implements an access control policy on the basis of an individual's need-to-know as opposed to mandatory controls which are driven by the classification ... of the information." [DOD] Discretionary access controls are complimentary to, rather than a replacement for, mandatory controls. While discretionary controls allow an individual to determine what user, or users, have access privilege to an object, it does not require that those users are trustworthy of that privilege. Likewise, while mandatory controls may grant access to an object to a trustworthy individual, it does not require that the individual has a need to know the object contents.

An example of a discretionary access control mechanism is the Access List [CS]. An Access List assigns to each object a list of access entries, each entry consisting of a user or group specification and a set of access privileges granted or denied to the specified user or group.

Section 2

The CAIS Discretionary Access Control Model

In a security controlled CAIS, an "object" is any CAIS node and a "subject" is any CAIS process node acting on the behalf of a given user.

2.1 Group Nodes

A CAIS user top-level node represents the individual user for discretionary access control. A named group is represented by a group node. A group node is simply a structural node with relationships defining each of the group's members. A group member may be either an individual user, a program, or another group.

A program is represented by the file node for the executable image of the program. A program is normally placed in groups only with other programs. For example, a program group may have as its members all compatible versions of the compiler, or a collection of tools designed to operate on a given type of data. Groups containing users may be called user groups, while groups containing programs are termed program groups.

Each group member is defined with either a primary relationship DOT or a secondary relationship POTENTIAL_MEMBER. The primary relationship DOT is used to define those members that are considered permanent members of a given group. It is used to create a hierarchy of groups and sub-groups by defining members of a group that are themselves groups. An individual user or program may not be the target of a primary DOT relationship emanating from a group node.

The secondary relationship POTENTIAL_MEMBER is used define those members that may dynamically acquire membership in the group. This act of acquiring membership is termed "adoption of group privileges" or simply "adoption". The phrase "potential member of a group" refers to any process, group, or program node that is the target of a POTENTIAL_MEMBER relationship from that group, or from any of that groups descendants. Figure 1 shows an example of a group structure defined with DOT and POTENTIAL_MEMBER relationships.

Certain group nodes can be accessed from a process node with the relation GROUP and a relationship key interpreted to be the group name or function. Two predefined user groups, ALL, which has all other groups (other than the group NONE) as its members, and NONE, which has no members, are provided. Thus the group containing the 'world' is accessed via the path 'GROUP(ALL) (fig. 2).

2.1.1 The Adopt Operation -- The adopt operation is used to acquire the privileges of a particular group. When a process adopts a particular group, a CAIS-controlled relationship, called ACTING_GROUP, is created from the process node to the group node. The phrase "acting member of a group" refers to any process node that has ACTING_GROUP relationship to that group or to one of the group's descendants. The phrase "acting group of a process" refers to any group node, or ancestor of any group node, that is the target of an ACTING_GROUP relationship from the process (fig 2). For a process to adopt a given group, an acting group of the process must be potential member of the given group. When any process node is created, it implicitly inherits the ACTING_GROUP relationships of its parent and adopts the file node for the program it is currently executing. When a root process is created, it implicitly adopts its current user node, as well as the file node for the program it is currently executing.

(A CAIS implementation may use group nodes to identify user groups for other purposes, such as job accounting, project or function identification. For example, a CAIS-controlled relationship attribute called PRIMARY_GROUP that is assigned to one of a process' ACTING_GROUP relationships and relates the process to its 'primary working group'. In general, group nodes are suitable for any user-defined relationship or attribute that would be meaningful to the group).

2.2 The Access Relation

An Access Relationship from an object to a group, user, or program node defines access privileges to the object that are either allowed or disallowed to the group, user, or program. Any object may have zero or more Access Relationships.

The Access Relation is called ACCESS and defines access privileges for a group, user, or program, and its relationship key distinguishes the group, user, or program name of the node that is the target of the relationship. Each Access Relationship may have Privilege Attributes which specify what access privileges are granted the group, user, or program.

Access Relationships may be established when an object is created, or at a later time. Implementations may provide a default set of Access Relationship that is established when an object is created. This set may include an Access Relationship which grants all access to the object's creator.

2.2.1 Privilege Attributes -- One Privilege Attribute is defined, called GRANT, which specifies what privileges are granted to the target group of the Access Relationship.

The Privilege Attribute value consists of a list of Privilege Specifications. Each Specification consists of a necessary privileges list, followed by a right-arrow, followed by a resulting privilege list. A list with one element may be replaced by the element alone. If the necessary privilege list is empty, the Privilege Specification may be replaced by the resulting privilege list alone.

The required and resulting privilege lists are lists of privilege names. A privilege name has the syntax of an Ada identifier. Privilege names may be user-defined, but certain privilege names have special significance to CAIS operations such as READ, WRITE, and CONTROL. The privilege name READ specifies the ability to perform read operations on the object, while the privilege name WRITE specifies the ability to perform write operations on the object. The privilege name CONTROL specifies the ability to perform access control functions on the object, such as modify the access control attributes. Other CAIS specific operations will be defined and may include CREATE, DELETE, APPEND, and UPDATE. (The intent here is not to define specific access privileges or procedure semantics, but rather to outline a general model for access control). The following are examples of privilege specifications:

```
(READ, WRITE, APPEND)

(COMPILE, CONTROL)

((EDIT, COMPILE)=>(READ, WRITE))

(READMAIL=>(READ, WRITE), SENDMAIL=>APPEND)
```

2.3 Access Control

Access control is performed for each operation requested on a given node. Each CAIS operation may require zero or more access privileges for each object that is accessed in the operation. For example, COPY_NODE requires READ privilege to the source node and WRITE privilege to the target node's parent.

Each required privilege for the object is compared to the process' privileges as defined by the object's Access Relationships. If the object has an Access Relationship to a group node that is an acting group of the subject and the relationship allows the privilege being checked, then the operation is allowed. Otherwise the operation is not allowed, and the operation is terminated by raising the exception ACCESS_VIOLATION.

For an Access Relationship to allow a privilege, the privilege must appear in a resulting privilege list in a GRANT attribute of the relationship and the privileges in the associated required privilege list must have been allowed.

(Likewise, a privilege attribute called DENY could be defined. in order to provided the ability to explicitly deny a right to a group or individual).

An Example

Consider, as an example, a CAIS implementation that supports two users, Smith and Jones.

1. Jones creates a file in his top-level node with protection specified by an ACCESS relationship to the group ALL with a GRANT attribute whose value is "READ". Because Smith has adopted the group TOOL_DEVELOPERS, which is a child of the group ALL, he is granted read access, but no other access, to the file (fig. 3).

2. Smith has, in his top-level node, a mail message file which is protected to allow read and write access only to Smith while he is executing a mail program, or to allow append access anyone while executing a mail program. This is done by creating an ACCESS relationship to Smith's top-level node granting the READMAIL privilege and an ACCESS relationship to the group ALL granting the SENDMAIL privilege. These privileges do not have special meaning to CAIS basic operations, and so do not grant any access to the object. However, an ACCESS relation is also created to the program group node MAIL_TOOLS that grants READ and WRITE privilege if READMAIL has been granted, and APPEND privilege when SENDMAIL has been granted. Smith creates a process, running the MAIL program, which causes that process to adopt the privileges of the MAIL program node, which has the affect of allowing the process READ and WRITE access to the mail file in Smith's top-level directory (fig. 4).

Conclusions

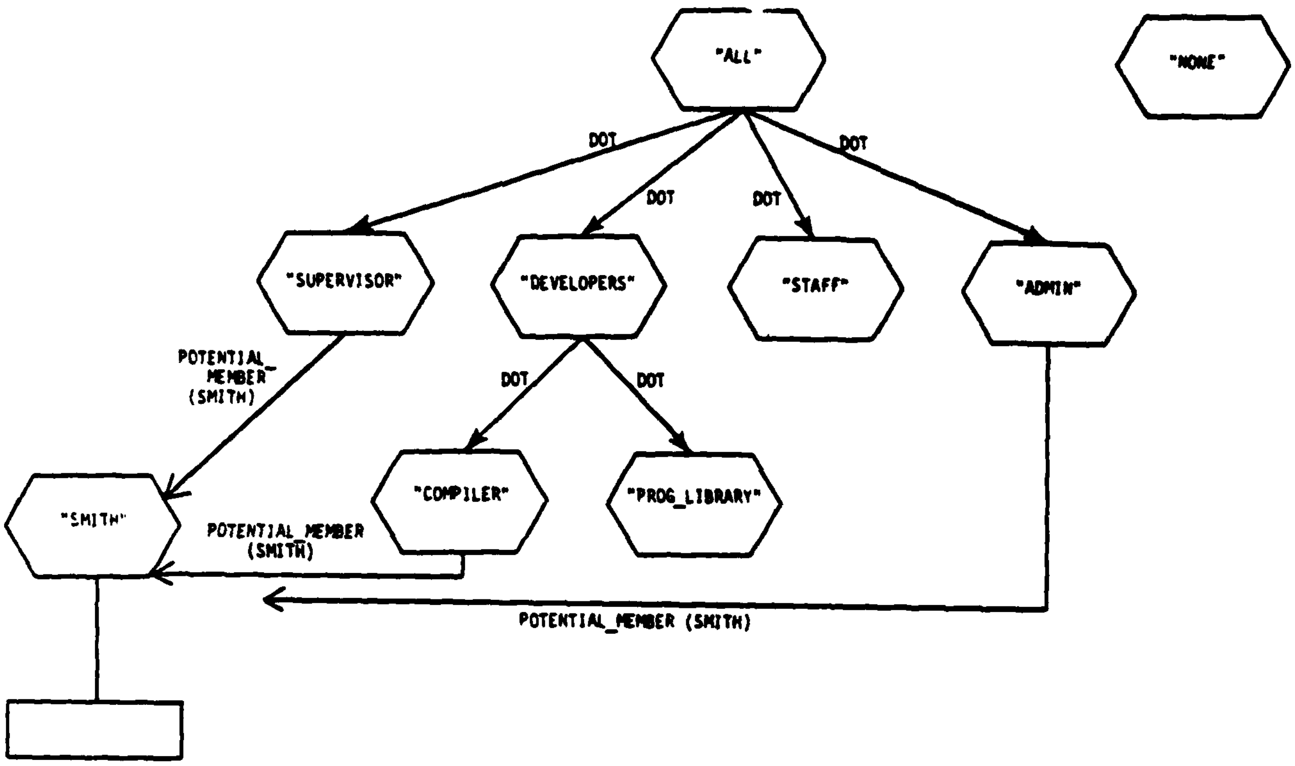
A discretionary access control mechanism for the Common APSE Interface Set has been described. The mechanism uses, as basic building blocks, the structures already provided by the CAIS and thus does not add significant complexity to the overall design of the CAIS. Only the basic structure of the mechanism has been defined and integrating the mechanism into the definition of the CAIS will involve further work in defining the specific access privileges required for CAIS basic operations, as well as defining any additional interfaces needed to support the mechanism.

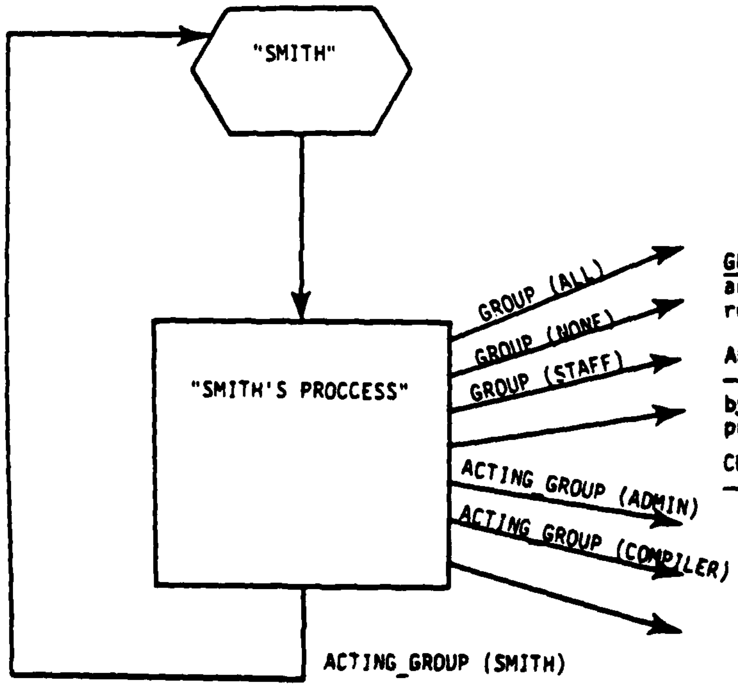
References

[CAIS] KIT/KITIA CAIS Working Group, Draft Specification of the Common APSE Interface Set (CAIS), AJPO, U.S. Department of Defense, 1983

[CS] Hsiao, David K., Kerr, Douglas S., and Madnick, Stuart E., Computer Security, Academic Press, Inc., New York, New York, 1979

[DOD] Computer Security Center, Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, Meade, Maryland, 1983

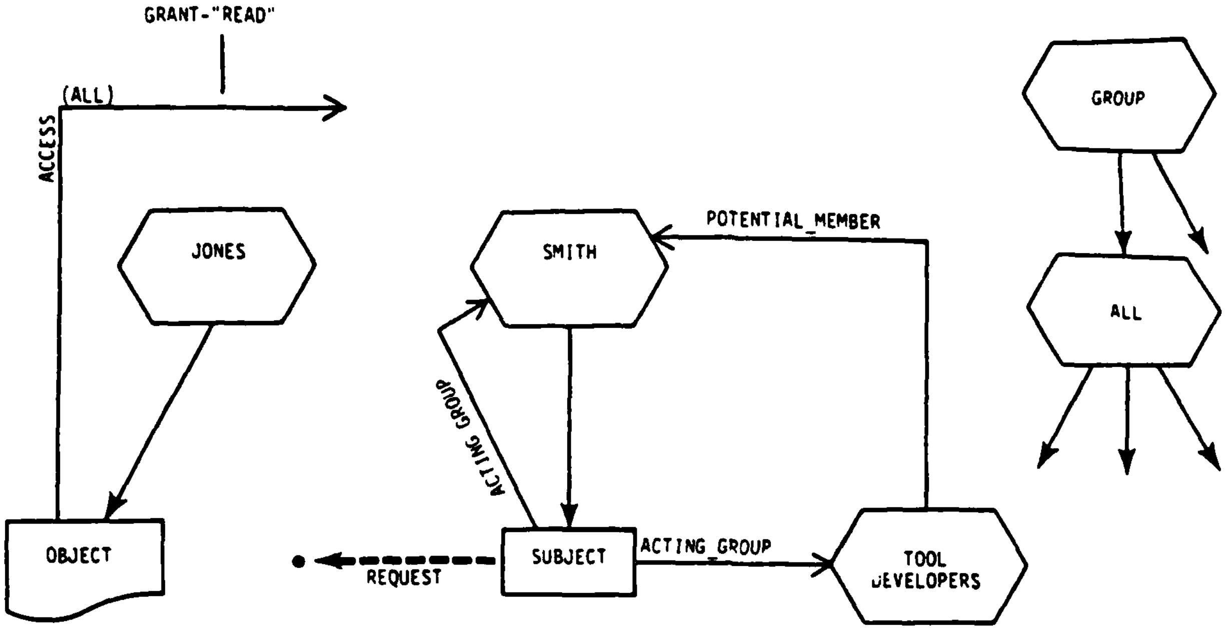




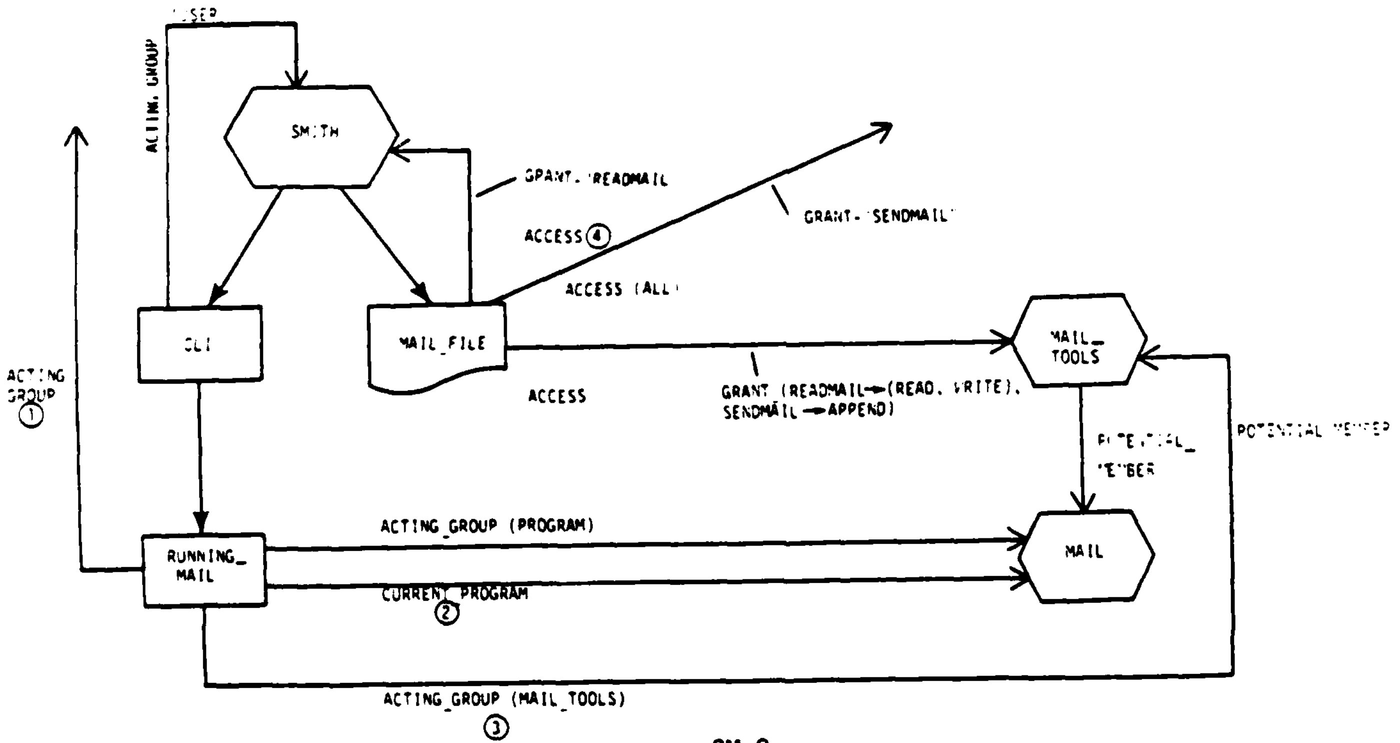
GROUP relationships provided for access to group nodes, like "USER" relationships.

ACTING_GROUP relationships are established by Adopt operation, which adopts the privileges of a group.

CURRENT_USER is implicitly adopted.



Smith Is Granted Access: Explicitly granted to group ALL which includes one of Smith's acting groups.



MANDATORY SECURITY MECHANISM FOR THE CAIS

KIT/KITIA CAIS WORKING GROUP

Abstract

Mandatory security protection and a mechanism for mandatory access controls for the Common APSE Interface Set [CAIS] is described.

Section 1
Introduction

In a trusted computer system, Security protection mechanisms provide controls over access by an individual, or a process acting on the behalf of an individual, known as the subject of the access, to a unit of information, known as the object of the access. Mandatory security protection provides access controls "based directly on a comparison of the individual's clearance or authorization for the information and the classification or sensitivity designation of the information being sought." [DOD]

A security classification may be either a hierarchical classification level or a non-hierarchical category. A hierarchical classification level is chosen from an ordered set of classifications levels and represents either the sensitivity of the object or the trustworthiness of the subject. In hierarchical classification, the reading of information flows downward towards less sensitive areas, while the creating of information flows upward towards more trustworthy individuals. A subject may obtain read-access to an object if the hierarchical classification of the subject is greater than or equal to that of the object. In turn, to obtain write-access to the object, a subject's hierarchical classification must be less than or equal to the hierarchical classification of the object.

A participant in the trusted computer system is assigned zero or more non-hierarchical categories which represent coexisting classifications applied to the subject or object. A subject may obtain read-access to an object if the set of non-hierarchical categories assigned to the subject contains each category assigned to the object. Likewise, a subject may obtain write-access to an object if each of the non-hierarchical categories assigned to the subject are included in the set of categories assigned to the object.

A subject must satisfy both hierarchical and non-hierarchical access controls to obtain access to an object.

Section 2

The CAIS Mandatory Access Control Model

In a trusted computer system, mandatory access controls are provided by assigning a label to each subject, typically a user or process, and object, usually a file or other unit of information. The label establishes the hierarchical classification and non-hierarchical categories for each object or subject. Access to an object is controlled at each operation request by a subject. If the subject, object, and type of operation meet mandatory access control requirements, the operation is allowed, otherwise the operation is denied.

Participants in the trusted computer base are CAIS nodes. Subjects requesting access are CIAS process nodes, while an object may be any CAIS node. Operations are CAIS operations and are classified as read, write, or read/write operations. Access is determined at the time the operation is requested, by comparing the classification of the subject with that of the object with respect to the type of operation.

2.1 Labeling

The labeling of participants is provided by CAIS-controlled node and relationship attributes. An attribute, called SUBJECT_CLASSIFICATION, is assigned to each process node and represents the node's classification as a subject. An attribute, called OBJECT_CLASSIFICATION, is assigned to each node and represents the node's classification as an object. The classification attribute is also duplicated as a relationship attribute for each relationship to the node. The value of the attribute is a parenthesized list containing two items, the hierarchical classification level and the non-hierarchical categories. The hierarchical classification is a keyword member of the ordered set of hierarchical classification keywords. The non-hierarchical categories is a list of zero or more keyword members of the set of non-hierarchical categories. For example, the following are possible participant classifications:

(TOP_SECRET, (MAIL_USER, OPERATOR, STAFF))

(UNCLASSIFIED, ())

(SECRET, (STAFF))

The following is a BNF for the value of a participant classification attribute:

```
    OBJECT_CLASSIFICATION  ::= CLASSIFICATION
    SUBJECT_CLASSIFICATION ::= CLASSIFICATION
    CLASSIFICATION ::= '(' HIERARCHICAL_CLASSIFICATION ','
                           NON_HIERARCHICAL_CATEGORIES ')'
    HIERARCHICAL_CLASSIFICATION ::= KEYWORD
    NON_HIERARCHICAL_CATEGORIES ::= '(' { KEYWORD { ',' KEYWORD } } ')'
    KEYWORD ::= IDENTIFIER
```

The hierarchical classification level set and the non-hierarchical category set are implementation-defined.

2.1.1 Labeling of Subjects -- When an individual gains entry to the APSE, a root process node is created at the top-level node of the user (see [CAIS]). When this process is created, it is assigned subject and object classification labels, based on the user's identification. The method by which these initial labels are assigned is not specified, however the labels "shall accurately represent security levels of the specific [users] with which they are associated." [DOD] When any process node is created, the creator may specify the classification attributes associated with the node. If no classification is specified, the classification is inherited from the creator. The assigned classification must adhere to the requirements for mandatory control over write operations.

2.1.2 Labeling of Objects -- When a non-process object is created, it is assigned an object classification label. The classification label may be specified in the create operation, or it may be inherited from the creator. The assigned classification must adhere to the requirements for mandatory control over write operations. When a file node is created, the classification label is assigned to it when the file is closed.

2.1.3 Labeling of Device Nodes -- Device nodes are not labeled on creation since devices are not dynamically created, like process and file nodes, and are generaly shared by users of different classification levels. Instead, the classification label cf a process opening a device node is assigned to the device node while it is open. This is similar to the concept of labeling the temporary file which is created, in some operating systems, on a terminal or printer when the device is opened.

There is a provision, however, for security of devices located in semi-secure areas. This is accomplished by assigning to the device node a range of allowable security classifications when the device is configured. Each device node has, in addition to the dynamically assigned classification labels, two additional

CAIS-controlled attributes. The attribute HIGHEST_CLASSIFICATION defines the highest allowable object classification label that may be assigned to the device node -- were a process opening the node labeled at a higher security level than that specified by HIGHEST_CLASSIFICATION, it would be possible to 'write down' since individuals of a lower classification may receive the information written.

The attribute LOWEST_CLASSIFICATION defines the lowest allowable object classification label that may be assigned to the device node -- were a process opening the node labeled at a lower security level than that specified by LOWEST_CLASSIFICATION, it would be possible to read information that is labeled with a higher classification, since individuals of a higher classification may send information to the device being read.

When a device node is opened the device inherits is security classification label from the process performing the open operation. If it is not possible to label the device node within the bounds of the attributes HIGHEST_CLASSIFICATION and LOWEST_CLASSIFICATION, the operation fails by raising the exception SECURITY_VIOLATION.

In addition, a CAIS implementation providing mandatory security provisions must provide appropriate labeling at the top and bottom of each page of human-readable output sent to devices such as terminals and printers.

2.2 Operation Types

Mandatory access controls require that operations be strictly typed as either read, write, or read/write. An operation is considered a read operation if any information flows from the object to the subject. An operation is considered a write operation if any information flows from the subject to the object. (This does not allow for indication of the success of a write operation). An operation is considered a read/write operation if information flows both ways. The subject must satisfy access controls for both read-access and write-access to obtain read/write-access to an object, i.e., the object and subject must have the same classification label. If access controls are not satisfied, the operation terminates by raising the exception SECURITY_VIOLATION. For example,

1. A file create operation is strictly a write operation to the destination file. However a process creating a file with a security classification that is higher than its own may not determine if the create successfully completed, or in any way access the file after it has been closed. After the file is closed, the process will have a relationship to the file node, but the EXIST function will raise SECURITY_VIOLATION

if called to determine if the node exists. Before the file is closed, it is locked for reading and writing and considered to be in the process's working memory. When the file is closed, it is released to the data base and given its classification.

2. A file open is at least a read operation. It is not possible for a process to write an existing file of a higher security classification, since opening the file (determining its existence) causes information to flow to the subject.

3. A process may create a process at a higher security classification, but may not determine when the created process terminates. The process may, however, delete the created process (remove its primary link to it).

4. A process may read a file that has a lower security classification, but may not lock the file in any way.

Section 3
An Example

Consider, as an example, a CAIS implementation in which has been defined the following classification sets:

Hierarchical:	(UNCLASSIFIED, SECRET, TOP_SECRET)
Non-hierarchical:	(A, B, C, D)

Also consider two typical users, Jones, who upon entry into the CAIS is assigned the classification (UNCLASSIFIED, ()), and Smith, who is assigned the classification (TOP_SECRET, (A, B).

1. Jones logs on to the CAIS and is created a root process which is assigned the classification "(UNCLASSIFIED, ())". Jones' terminal is assigned the same classification and displays on the screen "UNCLASSIFIED"

2. Smith sits down next to Jones and attempts to log on to the CAIS at a TOP_SECRET security level, but is prevented, since the Smith and Jones are sitting in an open terminal area and the device's HIGHEST_CLASSIFICATION attribute has the value "(UNCLASSIFIED, ())". Smith goes to his office, logs on to the CAIS and is created a root process, called SMITH_1, which is assigned the classification (TOP_SECRET, (A, B)). Smith's terminal is assigned the same classification and displays on the screen "TOP_SECRET, (A, B)".

3. Smith sets his current node to Jones's top-level directory and attempts to read some files. Smith reads a file

"TO_DO_LIST" labeled "(UNCLASSIFIED, ())". Smith attempts to read "COMMANDER_MEMO" labeled "(TOP_SECRET, (A, B, C, D))", but is denied, since SMITH_1's non-hierarchical categories do not include "C" or "D".

4. Jones creates a file labeled "(TOP_SECRET, (A, B))" called "FOR_SMITH" in his top-level directory. When it is closed the file becomes inaccessible to Jones, but readable by Smith.

5. Smith notices the file in Jones' top-level node and reads it. When done, Smith attempt to delete the file but is denied access, since this is a write operation to Jones' directory.

6. Smith attempts to print the file out on "'USER(PRINTERS).LP1" but is denied access since the device could not adopt the classification level of the file being printed. Its HIGHEST_CLASSIFICATION attribute has the value "(UNCLASSIFIED, ())".

7. Smith and Jones both log off the CAIS.

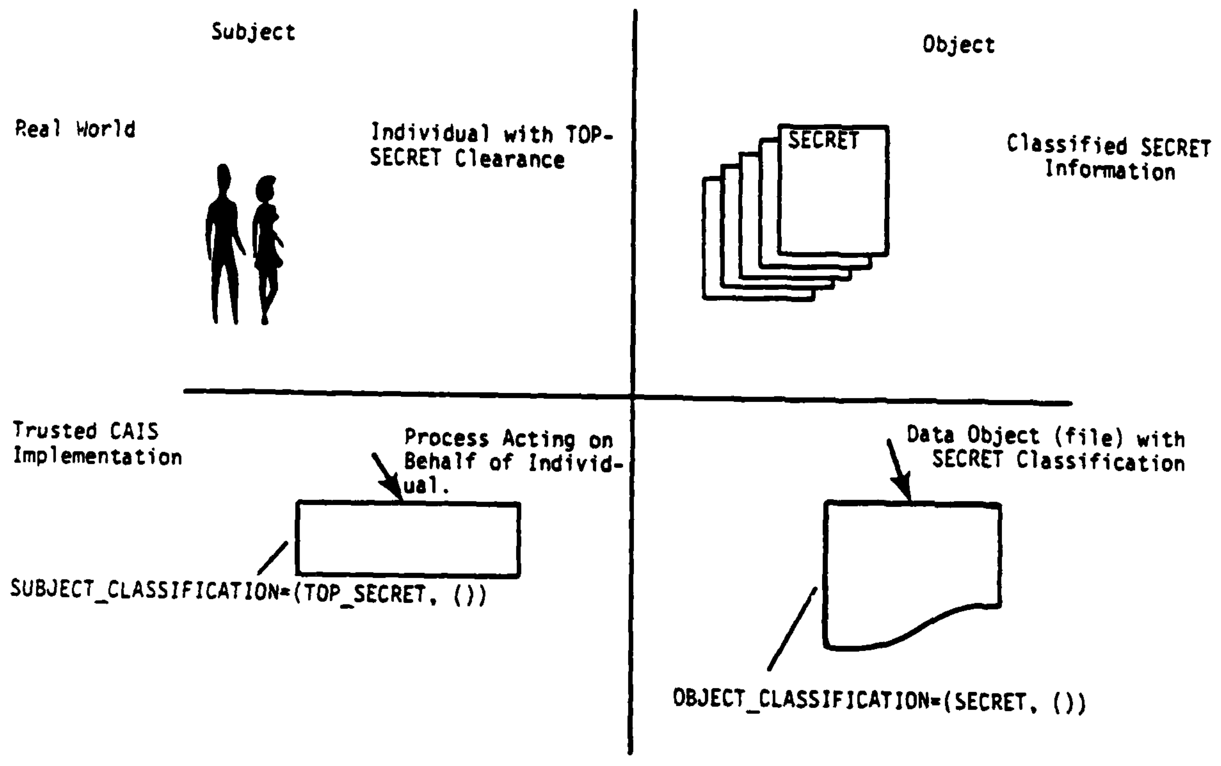
Conclusions

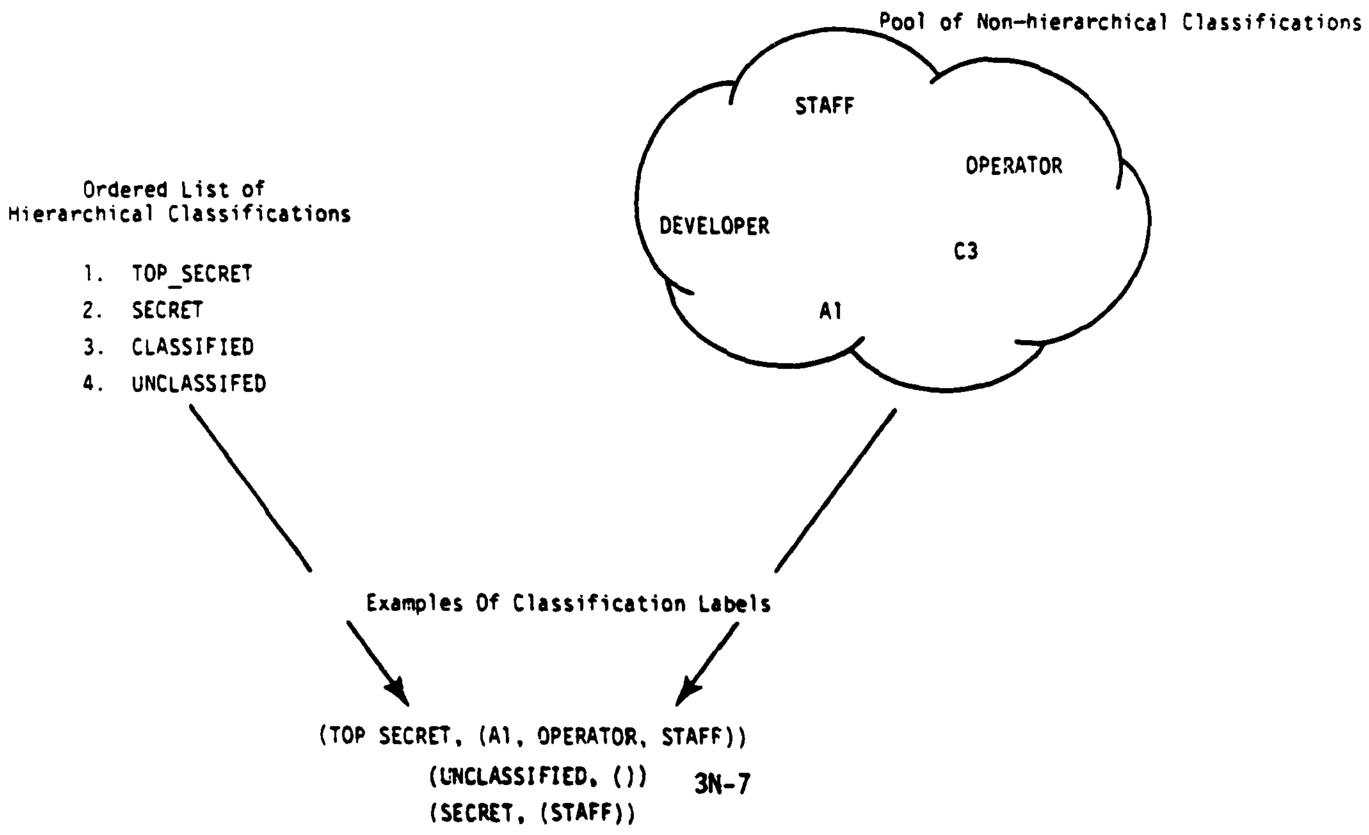
A mandatory access control mechanism for the Common APSE Interface Set has been described. Further work is required to define each basic CAIS operation type and to integrate the model into the CAIS specification.

References

[CAIS] KIT/KITIA CAIS Working Group, Draft Specification of the Common APSE Interface Set (CAIS), AJPO, U.S. Department of Defense, 1983

[DOD] Computer Security Center, Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, Meade, Maryland, 1983


Subject
Object
Real World
Individual with TOP-SECRET Clearance
SECRET
Classified SECRET Information
Trusted CAIS Implementation
Process Acting on Behalf of Individual.
Data Object (file) with SECRET Classification
SUBJECT_CLASSIFICATION=(TOP_SECRET, ())
OBJECT_CLASSIFICATION=(SECRET, ())


Pool of Non-hierarchical Classifications
STAFF
OPERATOR
DEVELOPER
C3
A1
Ordered List of Hierarchical Classifications
1. TOP_SECRET
2. SECRET
3. CLASSIFIED
4. UNCLASSIFED
Examples Of Classification Labels
(TOP SECRET, (A1, OPERATOR, STAFF))
(UNCLASSIFIED, ())
(SECRET, (STAFF))

CAIS PROGRAM TRANSPORTABILITY INTERFACE

Racwg

1. Introduction

The purpose of this paper is to examine the rationale for the Program Transportability Interface (PTI) of the Common APSE Interface Set (CAIS) [1]. The PTI has been proposed as one of three interface categories for specifying Interoperability and Transportability (I & T) requirements and criteria [2] that must be satisfied by the CAIS to facilitate the source level portability of Ada software development tools. The other two proposed interface categories are designated the Tool Transportability Interface and the Interoperability Interface.

The PTI encapsulates the fundamental portability requirements for a single Ada main program to execute consistently in different environments providing that the program only relies on those features that are described in the Standard Ada Language Reference Manual (ALRM) [3]. Achieving this level of portability is hindered by the implementor-defined options permitted in the ALRM. This permissivity can effect the execution behaviour of an Ada program when transported among dissimilar operating environments. Typically, this is most severe when the software and hardware processing capacities of the environments are substantially different. The specification of the requirements on the semantics of those options that are implementor-defined are necessary in order for the CAIS to include a comprehensive PTI. The preliminary CAIS was formulated without these requirements, however it clearly delineates the need to eliminate the implementation dependencies of the ALRM; e.g., 'While the semantics of the packages as specified are fully adhered to, the CAIS imposes additional requirements on those semantics that the LRM designates as being implementation-defined' (CAIS-4, 4.0 Para. 1).

Because an Ada main program may be expected to execute in both the software development and target execution environments, the implementation of the PTI represents the functionality that must exist in both the host and target environments. Normally this functionality is viewed as included in the Run-Time System (RTS) and Ada packages that are available from the Run-Time Support Library (RSL). Both the RTS and RSL use the underlying hardware or software.

An Ada program has an explicit interface to the RSL, e.g., Standard I/O, and an implicit interface to the RTS, e.g., tasking. Consequently, to achieve source level portability, a RTS and RSL must be available, in addition to a code generator, for the environments under which the program is to execute. A common assumption is that the RTS is totally dependent upon the environment and that the RSL is at least partially dependent upon the environment. The objective of the PTI is to minimize these dependencies by specifying appropriate requirements and criteria to be satisfied by the CAIS.

In order to assess the extent to which the PTI requirements and criteria may impact the I & T of future CAISs, the host and target Run-Time Environments (RTE) should be normalized. The remainder of this paper outlines a normalization strategy for the RTE that complements the rationale for the PTI.

2. Target RTE

The conventional perception of the target Ada RTE interfaces may be modeled through the following Ada syntax (NB- Ada syntax is used in lieu of interface graphics not suitable for word processing, and the syntax is not intended to convey an implementation technique) :

```
package ISA is ...  end;

with ISA;
package RTS is ...  end;

with ISA, RTS;
package RSL is ...  end;

with RTS, RSL;
procedure Target_Program is ...  end;
```

In this perception the RTE comprises the RTS and the RSL. The Ada main program depends upon interfaces provided by the RTS and RSL. The interface to the RTS supports the implementation of Ada semantics that are frequently included in an operating system. This interface is available to the code generator of the Ada compiler and is dependent upon the Instruction Set Architecture (ISA) of the target machine. The interface to the RSL conforms to the standard package specifications defined in the ALRM. Since it is a prevalent assumption that there is no target resident software outside of the RTE, the RSL interfaces to the ISA. In addition, because the RSL may be implemented using Ada, the RSL may also interface to the RTS.

As a result, in this model, the PTI requirements are restricted to those that are derived from the Ada standard package specifications, viz., Standard_IO. It is recognized that in many tactical embedded computer systems the functionality provided through the RSL is minimal thereby reducing PTI requirements.

3. Host RTE

The perception of the target RTE may be extended to model the RTE for a host. The RTE for a host, in this context, is assumed to enable multiple Ada main programs to share the underlying software and hardware processing resources, and in addition provides facilities required by an Ada software development environment. Although there is no accepted model for the host Ada RTE interfaces, the following Ada syntax presents a perception that is consistent with contemporary Ada Programming Support Environment (APSE) technology:

```
package HOST is ...  end;

with HOST;
package RTS is ...  end;
```

```
with HOST, RTS;
package KAPSE is ...  end;

with RTS, KAPSE;
package RSL is ...  end;

with RTS, RSL;
procedure Host_Program is ...  end;

with RTS, RSL, KAPSE;
procedure APSE_Tool is ...  end;
```

In this perception the RTE comprises the RTS, RSL, and Kernel APSE (KAPSE). The significant extension to the model for the target Ada RTE is the inclusion of the KAPSE. The interface to the KAPSE supplies operating system type services required for portable APSE tools, that are outside of those provided by Ada semantics, through its implementation-dependent interface to the underlying software, firmware, or hardware. Because the host environment is a target environment for Ada programs, it provides the same interfaces to the RTS and RSL as the target model. However, the functionality of the RSL is extended to supply additional tool services. The RSL comprises Ada packages that are portable to the same extent as an APSE tool. This is achieved by relocating RSL host dependent services to the KAPSE. These services are specified in the KAPSE interface and are not precluded from being implemented in Ada. As a consequence, the KAPSE may use the RTS interface.

The PTI requirements for the host RTE are changed by the extensions to the target RTE model because of the portability envisaged for the RSL. For example, the PTI may now include more primitive I/O services, in addition to those encapsulated by Standard_IO.

4. Normalized RTE

The potential increase in portability of the RSL in the host RTE derived through the functionality of the KAPSE offers the opportunity to explore a normalized RTE applicable to both host and target RTEs. The key to the proposed normalization strategy are the requirements placed upon the PTI.

The degree of portability achieved for the RSL is dependent upon identifying canonical operating system type services that can be encapsulated by the KAPSE for reconciliation with the processing characteristics of the host and target. These services are included as requirements for the PTI. Similar requirements may now be developed for the RTS. These requirements are predicated on the feasibility of an Ada support kernel. This support kernel would allow part of the RTS and the KAPSE to be implemented in Ada. A possible candidate that appears suited to this separation policy are the run-time task management functions [4]. For example, the rendezvous semantics may be implemented using sequential Ada relying on the support kernel to share the processing resources for execution. Ada programs that enclose tasks may now interface to standardized task management functions of the RTS. This interface would be amenable to an Ada subprogram specification, but would be materialized into an executable form by the code generator and is therefore not included in the PTI. An expected advantage of this decomposition is that a consistent model for code execution is available to the Tool Transportability Interface through the KAPSE for the execution of multiple main programs. The KAPSE is provided the same execution services for process management as the RTS is provided for implementing task management. Consequently, the PTI requirements are dependent upon the support kernel interface.

4.1 Normalized Target RTE

The adaptation of the target Ada RTE interfaces into the normalized form using the notion of a Target Ada Support Kernel (TASK) [5] may be modeled through the following Ada syntax:

```
package ISA is ...  end;

with ISA;
package TASK is ...  end;

with TASK;
package RTS is ...  end;

with TASK, RTS;
package RSL is ...  end;

with TASK, RTS, RSL;
procedure Target_Program is ...  end;
```

The RTE comprises the TASK, RTS, and RSL. The TASK is analogous to an executive system for the target that services the PTI and the run-time support for Ada programs that is not included in the PTI. The TASK is the only component that interfaces to the ISA. In addition, the TASK may include the functionality to support the distributed execution of Ada tasks, where distributed connotes both shared and separated processing resources. While more general distributed execution of Ada programs is the topic of current research [6], an important requirement of the PTI is for an Ada program enclosing tasks to execute consistently in either a distributed or non-distributed environment.

4.2 Normalized Host RTE

The adaptation of the host Ada RTE interfaces into the normalized form uses an identical notion of a Target Ada Support Kernel (TASK) and may be modeled through the following Ada syntax:

```
package HOST is ...  end;

with HOST;
package TASK is ...  end;
```

```
with HOST, TASK;
package KAPSE is ...  end;

with TASK;
package RTS is ...  end;

with TASK, KAPSE, RTS;
package RSL is ...  end;

with TASK, KAPSE, RTS, RSL;
procedure APSE_Tool is ...  end;
```

The RTE comprises the TASK, KAPSE, RTS, and RSL. The functionality of the TASK is consistent with the host/target isomorphism of the PTI. As a result, the PTI requirements for the host RTE are identical to those of the target RTE. Therefore, any distributed behaviour of a target program should be supported on the host either logically or physically to enhance program testing in the APSE.

5. Conclusions

The proposed normalization of the RTE for hosts and targets presents an argument for a set of PTI requirements that can enhance the transportability of a single program, not only among APSEs but also between an APSE and a target. The PTI addresses several requirements that have been categorized as deferred topics by the preliminary CAIS; e.g., the support of non-software development (target) environments and distributed execution environments. A coherent approach to the specification of PTI requirements for these topics is facilitated by the normalized RTE. This normalization introduces the functionality of both a KAPSE and a TASK for the host RTE. The PTI comprises services that are performed through the TASK, KAPSE, RTS, and RSL. However, not all these services can be implemented as Ada subprograms and may therefore be outside the the scope of a near-term CAIS. The requirements for these services raise the following issues that have a potential impact on the transportability of a program:

a) simplification of retargeting Ada compilers
b) utilization of available processing capacity
c) isolation of an Ada Virtual Machine
d) adaptation of a program for distributed execution
e) adaptation of a program for multilevel secure execution

The retargeting of the Ada Compiler is essential if an existing Ada program is to execute in a new environment. Retargeting can at a minimum involve implementing a new code generator and a RTS. Any reduction in the magnitude of this effort will expedite transportation of a program, whereas a costly effort may preclude retargeting the Compiler and thereby prevent the execution of the program in the new environment. The normalized RTE attempts to simplify retargeting by requiring that the conventional RTS and RSL be partially transportable through the use of the PTI serviced by the TASK.

The efficacy of transporting a program may depend upon whether the program exhibits reasonable performance characteristics in different execution environments. The normalized RTE presents interface requirements that separate Ada semantic interfaces from those interfaces that are necessary to ensure efficient use of the available processing capacity. The ability to effectively test a program in the host depends on the degree of capacity transparency achieved by the PTI.

The separation of interfaces and the accompanying functional decomposition provide abstraction levels for an Ada Virtual Machine. These levels may be used to define requirements for future firmware and hardware.

The adaptation of a program to logically or physically distributed environments is dependent upon the PTI. The need to execute a program in loosely-coupled target environments has been identified [7], where testing facilities are required on both the host and target.

The formulation of transportability requirements for multilevel secure execution may become a dominant issue in developing future CAISs. While multilevel security execution is not currently mandated for APSEs it is a requirement of target software [7]. Therefore, achieving multilevel secure execution in a target RTE should lead to the same capability in the host RTE through the the expected properties of normalization. The I & T requirements for distributed and secure execution may be addressed initially through the PTI that is not constrained by the host environment.

6. References

[1] Preliminary Specification of the Common APSE Interface Set (CAIS) - KAPSE Interface Team CAIS Working Group, 1983-08-08.

[2] Requirements and Criteria Working Group Workshop - San Diego, 1983-07-12.

[3] Ada Programming Language - ANSI/MIL-STD-1815A, 1983-01-22.

[4] CSC/NASA Real-Time Ada 8086.

[5] Program Invocation and Control - KIT Public Report Vol. 2, 1982-10-28.

[6] Distributed Ada Project Report - Honeywell SRC, 1983-01.

[7] System Specification for Ada Language System/Navy - NAVSEA, 1982-10-15.

Position Paper: The Need for a CAIS Operational Semantics

R. S. Freedman KITIA-WG1/COLMPWG

The specification of CAIS semantics is necessary for the determination of KAPSE interface compliance. Part of the problem in formulating a detailed specification of CAIS semantics is related to our inability to develop a formal model for program interoperability and transportability. The requirements and criteria for interoperability and transportability directly impact the CAIS semantics, since implementations that comply to the CAIS specification are supposed to permit and support interoperable and transportable Ada programs.

The existing approaches to specifying CAIS semantics have been based on the creation of abstract models. The several notations that have been provided with these models have ranged from highly formal (denotational semantics), informal (similar to the "structured" English found in the Ada RM), to somewhere in between (the abstract machine approach). All of these abstract models have the property of being independent of a particular implementation. This is an important property in view of the fact that the CAIS is supposed to be a "machine independent" specification. However, because of the disagreements over the exact support needed for interoperability and transportability, the specification of CAIS semantics in terms of an abstract model may be a bit premature.

It is advocated that an operational semantics for the CAIS be developed, in parallel to the existing abstract approaches. The development of a CAIS operational semantics would have the advantage of rapidly revealing possible ambiguities in the CAIS specifications. A secondary advantage would be in the ability of rapidly prototyping and testing different implementation strategies, in order to verify concept feasibility associated with program interoperability and transportability.

In particular, interoperability and transportability can also be given an operational definition associated with the performance and behavior of an Ada program that utilizes the CAIS operational specification. A program that uses the CAIS packages and whose actual behavior corresponds exactly to its predicted behavior can be considered to be (operationally) a transportable and interoperable program.

The development of an operational semantics is not the only method available for specifying CAIS semantics; however, it may be the only formal method available that may help the CAIS designers test their ideas. The operational approach also has a good legacy in the Ada program: the NYU Ada/ED project showed the advantages of building an operational specification of the Ada language even before the language design was frozen. An operational semantics for the CAIS would have similar benefits.

TOWARD A FORMAL SEMANTICS FOR THE CAIS

Larry Yelowitz

Ford Aerospace & Communications Corp.

1. Introduction

This paper is the first in what is hoped will be a series culminating in a formal semantics for all (or most) of the CAIS. One of the ubiquitous comments received from the public review of CAIS 1.1 is the need for a semantics. There are a variety of methods for presenting a formal semantics, eg, axiomatic, algebraic, denotational, etc. There is no escaping the fact that some degree of mathematical maturity is required to comprehend any formal semantics. It is my feeling, however, that the semantics presented here, which is a variant of the axiomatic method, is the most comprehensible to the largest set of serious readers of the CAIS.

1.1 Issues Involving Involved in Presenting Semantics of a System

The following issues are involved in presenting semantics of a system:

1. Notation to describe states of the system. Desirable properties of the notation are that it have a clear syntax and semantics, and thus be amenable to machine checking for errors.

2. Axioms and proof rules for proving theorems in the system.

3. Specific system dependent axioms which characterize the system under investigation. Sometimes this set of axioms is known as "invariants" (since they are required to be true in the initial state, and to remain true under state changes). Sometimes this set of axioms is called "law and order" assertions or invariants.

4. Mechanisms by which state changes may occur. Notation is needed for specifying the new state in terms of the old. There are advantages in specifying the state transitions "nonprocedurally", ie, the new state is given directly in terms of the old, rather than by a step-by-step procedure which visits numerous intermediate states along the way.

5. Mechanisms by which the notation describing state transitions may be mapped into candidate theorems in the notation describing states. Such mechanisms are generally called "verification condition generators". The candidate theorems have the property that if they are valid, then the invariants really are invariant under the state transitions.

6. Proof that the candidate theorems are valid.

1.2 A Trivial Example

We first present a trivial example to exemplify the six aspects.

1. The notation for describing states is high school level algebra over the integers. (We skip a formal presentation of this notation).

2. The axioms and proof rules are the familiar associative, distributive, commutative "laws", etc., plus various axioms on the equality relation (eg, "equals added to equals are equal").

3. The law and order invariant is: y = 3*x, where x and y are named entities in the system. (We assume that x and y are initialized so this invariant is true).

4. The mechanism by which the state may change is given as follows:

   ```
   procedure UPDATE;
   out y = in y+3;
   out x = in x+1;
   ```

   This is a nonprocedural description of the effect that UPDATE must have on the state, ie, the value of y in the new state must equal the value of y in the old state plus 3, and the value of x in the new state must be 1 greater than the value of x in the old state. It is tempting (especially for programmers in languages in which the assignment operator and equality predicate are the same symbol, eg, Fortran) to view this description as a procedure. It is not. This description simply characterizes the net effect that any procedural implementation must satisfy. There are an infinite number of procedures which satisfy these specifications.

5. The verification condition generator must transform the semantics of UPDATE to candidate theorems in the problem domain of integers. For example, the problem

domain does not include reserved words "in" and "out", so the vcg must somehow eliminate them, while preserving the semantics. Various methods of "path analysis" have been proposed by researchers in program verification. Using "forward" path analysis, the candidate theorem would be:

A. y=3*x;

B. x'=x+1;

C. y'=y+3;

D. Based on the above hypotheses, prove: y'=3*x'.

6. The proof that the candidate theorem is valid can be given simply based on axioms for equality and the distributive law.

1.3 How the Semantics Relate to the CAIS

In this subsection we discuss briefly the relationship of each of the six aspects of semantics to the CAIS, and indicate in which section of this paper it is dealt with.

1. Notation for describing states: states in the CAIS are described in terms of a directed graph model, with nodes, arcs, arc labels, paths, trees, connectivity, and related notions. This notation is presented in Section 2. Section 2.1 presents notation dealing with the digraph notions just mentioned. Section 2.2 presents additional notation to particularize the discussion to the CAIS, eg, by presenting notions such as users and top level nodes.

2. Axioms and proof rules for proving theorems in the domain of digraphs: This version of the paper is weak in this area. We rely on the reader to have some intuitive feeling for such proof rules. For example: given a suitable axiomatization, the following theorem is provable: If there is a path from node x to node y, and a path from node y to node x, then there is a cycle.

3. The invariants which give the CAIS its specific properties are given in Section 3. The notation for stating the invariants uses first order predicate calculus, plus the digraph notation. My feeling is that the invariants form a good basis for CAIS 1.1, sections 3, 4, and 5 (CAIS Node Model, CAIS Structural Nodes, and CAIS File Nodes). That is, although

additional assertions are undoubtedly going to be required, they will be of the same genre as the assertions given in this version of the paper. On the other hand, I expect that additional assertions of a rather different genre will be needed for the semantics of the other sections of CAIS 1.1 (CAIS Process Nodes, CAIS Device Nodes, and CAIS Utilities).

4. State transitions are specified using notation similar to that in the trivial example above. We borrow heavily from the language ANNA [Draft: Reference Manual for ANNA: A Language for Annotating Ada Programs, Luckham et. al]. Section 4 provides formal state transitions for Copy_Node, Copy_Tree, and Rename, from package CAIS_NODE_MANAGEMENT.

5. The verification condition generator is not dealt with in this version of the paper.

6. Proofs that the law and order assertions remain invariant is given casually in this version of the paper.

1.4 An Explanation of Notation

1. We use "forall" and "exist" as the predicate calculus quantifiers.

2. INSET refers to the two place predicate, normally written as an epsilon, dealing with set membership. Eg, x INSET A says that the element x is a member of the set A.

3. UNION refers to set union.

4. If A is a set, 2**A denotes the power set of A, ie, the set of all subsets of A.

5. In presenting the semantics of procedures, we first give a list of assertions describing the exceptions that may be raised, in the form:

   ```
   raise exception_name => boolean_condtion;
   ```

   The symbol "=>" is simply a lexical delimiter, NOT the implication sign. The semantics is that if the given exception_name is propagated by the procedure, then the boolean_condition is true at the point in time at which the exception handler receives the propagation.

   After the assertions characterizing which exceptions may be propagated have been given, there are a series of assertions characterizing the new state in terms of the old. Although these assertions are necessarily listed in some sequence, the semantics is that the new state is derived instantaneously from the old, ie, all the effects become true in one fell swoop. (This is analogous to concurrent assignment, as opposed to sequential assignment statements in some programming languages).

6. Logical implication is given by the symbol "->".

2. Directed Graph Properties of the CAIS

2.1 Underlying Notation and Definitions

1. Nodes is a dynamic set of nodes.

2. Arcs is a dynamic set of arcs, ie, ARCS SUBSET Nodes x Nodes. A given arc may be referred to either as (n1,n2), or more abstractly simply as, eg, "a". In the latter case, we establish correspondence with the former notation by referring to SOURCE(a) and SINK(a) to represent the appropriate nodes.

3. Labels is a set of labels that are associated with Arcs. There is a function Label: Arcs -> Labels providing the label associated with a given arc. Labels is an abstraction of the relation name, or the (relation name, relation key) pair associated with each arc in the CAIS.

4. Labels* is a sequence of elements from Labels, including the null sequence. For a given element, seq, from Labels*, there are the following definitions:

   A. Length(seq) is an integer >= 0, providing the length. In later versions of this paper, Length can be defined formally.

   B. seq(i) refers to the ith element of the seq. seq(i..j) refers to the "slice" of seq consisting of elements i through j inclusive; undefined if j<i. The following items can be defined in terms of seq(i) or seq(i..j), but it is sometimes helpful to have the following notation also.

   C. First(seq) is the first element of seq, provided Length(seq) >0. First is undefined otherwise.

D. Tail(seq) is seq with the first element deleted; undefined if Length(seq) =0.

E. Last(seq) is the last element of seq; undefined if Length(seq) = 0.

F. AllButLast(seq) is seq with the last element deleted; undefined if Length(seq) = 0.

5. OutgoingArcs: Nodes -> 2**Arcs is a function providing for each node, a set of outgoing arcs from that node. That is, a INSET OutgoingArcs(n) iff exist n1. INSET Nodes st (n,n1) INSET Arcs and SOURCE(a) = n and SINK(a) = n1.

6. IncomingArcs: Nodes -> 2**Arcs is a function providng for each node, a set of incoming arcs into that node.

7. IsPrimaryArc: Arcs -> {true,false} is a predicate partitioning the set of arcs into primary and nonprimary arcs.

8. The following functional property must hold regarding arc labels: (forall n INSET Nodes) (forall a1,a2 INSET Arcs) (a1 INSET OutgoingArcs(n) and a2 INSET OutgoingArcs(n) and a1 /= a2 -> Label(a1) /= Label(a2)). That is, distinct arcs emanating from the same node must have distinct labels.

9. TargetNode: Nodes x Labels -> Nodes =def TargetNode(x,R) = y iff (x,y) INSET Arcs and Label((x,y)) = R. Undefined if there is no node y with this property.

10. For a given x,y INSET Nodes, R INSET Labels, we have the following definitions:

A. P(x,R,y) =def (x,y) INSET Arcs, IsPrimary((x,y)), and Label((x,y)) = R.

B. S(x,R,y) =def (x,y) INSET Arcs, not IsPrimary((x,y)), and Label((x,y)) = R.

C. D(x,R,y) =def (x,y) INSET Arcs, and Label((x,y)) = R.

In other words P(x,R,y) means there is a primary arc from x to y with label R; S(x,R,y) means there is a nonprimary arc (ie, Secondary arc) from x to y with label R, and D(x,R,y) means there is an arc, either primary or secondary (ie, Don't care) from x to y with label R.

11. Having defined P(), S(), and D(), we now define their closure as follows. For a given x,y INSET Nodes, seq INSET Labels*:

P*(x,seq,y) =def (Length(seq) = 0 and x=y)

or

(length(seq) >0 and exist z INSET Nodes

(P(x, first(seq), z) and P*(z,tail(seq),y).

12. S*(x,seq,y) =def (Length(seq) = 0 and x=y)

or

(length(seq) >0 and exist z INSET Nodes

(S(x, first(seq), z) and S*(z,tail(seq),y).

13. D*(x,seq,y) =def (Length(seq) = 0 and x=y)

or

(length(seq) >0 and exist z INSET Nodes

(D(x, first(seq), z) and D*(z,tail(seq),y).

14. There is a specially designated node, SystemRootNode. (This may be a fiction of the formal semantics, without a counterpart in the actual CAIS. The purpose is to provide logical notation for stating various properties, such as the tree property given next).

15. The nodes form a tree with respect to primary arcs. We have enough mechanism now to state this property f~rmally, as the following two subproperties:

A. (There is a primary path from SystemRootNode to all nodes). (forall n INSET Nodes)(exist seq INSET Labels*) P*(SystemRootNode,seq,n).

B. (The primary path to a given node is unique) (for all n1, n2 INSET Nodes)(forall seq1,seq2 INSET Labels*) (P*(n1,seq1,n2) and P*(n1,seq2,n2) -> seq1=seq2).

16. TreeSet: Nodes -> 2**Nodes is defined as follows:

TreeSet(n) =def {n1|exist seq INSET Labels*

P*(n,seq,n1)†

Thus, for a given node n, TreeSet(n) is the set of all nodes reachable from n following only primary arcs.

2.2 Notation and Definitions Specific to the CAIS

1. There is the following partitioning of Nodes into four subsets: ProcessNodes, StructuralNodes, FileNodes, and DeviceNodes. There is a subset of ProcessNodes known as RootProcessNodes. There is a subset of nodes known as TopLevelNodes.

2. There is a set of users, known as Users. For simplicity in this version of the formal semantics, the following assumptions are made:

   A. Users is a fixed static set.

   B. A given user may be logged on to the CAIS at most once at any time. (We will deal formally with the notion of "logged on" below).

3. TopLevelNode: Users -> TopLevelNodes is a function providing the top level node associated with a given user.

4. UserAssociatedWith : TopLevelNodes -> Users is the inverse function of TopLevelNodes. Below we generalize the domain of UserAssociatedWith.

5. NodeStatus: Nodes -> §Opened,Closed,Deleted†.

6. LoggedIn: Users -> §true,false†.

7. LoggedInUsers =def §u¶u INSET Users and LoggedIn(u)†.

8. IsTopLevelNode: Nodes -> §true,false†. --true only for top level nodes.

9. UserName: Users -> string.

10. JobName: Users -> string UNION §undefined†. JobName for a given user is defined iff the user is logged in, ie,

    (forall u INSET Users)(JobName(u) INSET string <-> LoggedIn(u)).

11. RootProcessNode: LoggedInUsers -> RootProcessNodes is a function providing the RootProcessNode for a given logged in user.

12. TreeRoot: ProcessNodes - SystemProcessNode -> RootProcessNodes is defined as follows: TreeRoot(x) = x if x INSET RootProcessNodes. For x not in RootProcessNodes, TreeRoot(x) =def y, where y has the following properties:

    A. y INSET RootProcessNodes.

    B. x INSET TreeSet(y), ie, there is a primary path from y to x.

    For TreeRoot to be well defined, we need an assertion that there is always a unique y satisfying these two properties. This assertion is given below.

3. Assertions Characterizing the CAIS

The following assertions are intended to hold true in all states of a CAIS.

1. (forall u INSET LoggedInUsers)
   S(TopLevelNode(u),(JOB,Jobname(u)),RootProcessNode(u)).

   That is starting from the top level node of a given logged in user, and following the (JOB, jobname) secondary arc, the root process node for that user is reached.

    A. (forall x,y INSET Nodes)(x/=y -> TargetNode(x,JOB) /= TargetNode(y,JOB)). That is, there is no way to reach a single RootProcessNode from two distinct nodes by following the JOB label.

    B. (forall x INSET RootProcessNodes)(exist y INSET TopLevelNodes)
       (exist name INSET string)
       S(y,(JOB,name),x).

       That is, there is a one-to-one mapping between root process nodes, and top level nodes. The CAIS has the label JOB to get from the latter to the former, but does not have a special named relation to get from the former to the latter. For the moment, let us refer to y as JOB_INVERSE(x).

We can now generalize the domain of the function UserAssociatedWith which was given above on the domain TopLevelNodes. We extend the domain to include ProcessNodes. The definition is that UserAssociatedWith a root process node x equals UserAssociatedWith(JOB_INVERSE(x)). For a process node x *not* in the *set RootProcessNodes*, UserAssociatedWith(x) =def UserAssociatedWith(TreeRoot(x)).

2. (forall U INSET LoggedInUsers)
 S(SystemProcessNode,
 (USER,Username(u)),TopLevelNode(u)).

 That is, starting from the (ghost?) SystemProcessNode, it is possible to reach the top level node of any given logged in user, by following the secondary arc (USER, username).

3. (forall p INSET ProcessNodes - RootProcessNodes)
 S(p,CURRENT_JOB,TreeRoot(p).

 That is, in any job tree, every node (other than the root) points to the root via the secondary arc CURRENT_JOB.

4. (forall p INSET ProcessNodes - SystemProcessNode)
 LoggedIn(UserAssociatedWith(p)).

 That is, for every process node, the associated user is logged in.

5. (forall p INSET ProcessNodes - SystemProcessNode)
 S(p,
 CURRENT_USER,TopLevelNode(UserAssociatedWith(p)).

 That is, from any process node, by following the CURRENT_USER secondary arc, one reaches the top level node of the associated user.

6. (forall x,y INSET Nodes, R INSET Label)
 P(x,R,y) -> S(y,PARENT,x).

 That is, *if there is a primary relationship from* x to y, then there is a secondary PARENT relationship from y to x.

7. (forall x,y INSET Nodes)
 S(x,PARENT,y) -> (exist R INSET Label)P(y,R,x).

 This is the converse of the preceding assertion, and states that there can only be a secondary PARENT relationship from x to y if there is a primary relationship from y to x.

8. The nodes form a tree with respect to primary arcs. (The formal assertion was given above).

9. (forall n INSET Nodes)(exist seq INSET Label*)(exist n1 INSET Nodes)
```
    IsTopLevelNode(n1) and
    (forall i INSET 1..Length(seq)) seq(i) = PARENT and
    S*(n,seq,n1).
```

   That is, "every node can be traced back to its top level node by recursively following PARENT relationships" [CAIS 1.1, p 3-2].

10. (forall n, n1 satisfying the previous assertion)
```
    (exist seq INSET Label*)
    P*(n1,seq,n).
```

   That is, "the path obtained by inverting this chain is the unique primary path to the node." [CAIS 1.1, p3-2].

11. There is a functional property with respect to arc labels on outgoing arcs from a given node. (The formal asserion was given above).

12. (forall p INSET ProcessNodes - RootProcessNodes)
```
    (exist unique r INSET RootProcessNode)
    p INSET TreeSet(r).
```

   This assertion is needed for the definition of TreeRoot given above to be well-formed.

4. Semantics of Selected Procedures

In this section is provided semantics of certain procedures, plus an informal analysis of whether the law and order assertions remain invariant after the procedure has caused its effect, given that the assertions are true prior to the procedure call. The semantics would normally be given by providing pre- and postconditions, characterizing the state before and after the procedure call. In these procedures, there is no precondition (or more precisely, the precondition defaults to TRUE). The postconditions also indicate various exceptions that may be raised, and the

state of the system at the time such exceptions are
propagated.

4.1 The Procedure Copy_Node

```
procedure Copy_Node(x,y: in Nodes);


  1.   raise USE_ERROR => x INSET ProcessNodes UNION
      DeviceNodes;

  2.   raise USE_ERROR => y INSET ProcessNodes UNION
      DeviceNodes;
        --if either x or y is a process or a device node, the exception
        --USE_ERROR is returned.

  3.   raise USE_ERROR => (exist z INSET Nodes)(exist R
      INSET Labels)P(x,R,z);
       --if there is any primary relationship emanating out of x, raise USE_ERROR.

       --Assuming that none of the above exceptions is propagated, the following
        --are the  effects of this procedure.


  4.    (forall z INSET Nodes)(forall R INSET Labels)
         ( (z/=x and in S(x,R,z) -> out S(y,R,z));
           --all secondary relationships out of x to a node other than x
           --have been copied to y.


  5.   (forall R INSET Labels)
        (in S(x,R,x) -> out S(y,R,y));
         -- if there were one or more reflexive arcs from x
       to itself, then
         --these arcs have been copied from y to itself, with
       the same labels.


  6.    (out CONTENTS(y) = in CONTENTS(x))
          and
        (out ATTRIBUTES(y) = in ATTRIBUTES(x))
         --contents and attributes have been copied from x to
       y.


  7.    (in S(y,R,z) and not in S(x,R,z) -> not out S(y,R,z))
        --This one might be controversial.  It says that all
       secondary
        --relations that originally emanated from y have been
       deleted,
```

--unless they are (re)included as part of the Copy operation itself.

8. --An alternative might be to make y a newly created node, in which
   --case there would not be any arcs coming from y in the precondition state.
   --In this case the last effect written above could be replaced by
   -- the following:
   y not INSET in Nodes and y INSET out Nodes;

4.1.1 Analysis of Copy Node Semantics In this section we informally analyze whether the law and order assertions given above remain invariant under the semantics of Copy_Node. If y is a newly created node, then there is one problem area: invariant 7 may be violated in the new state, since it is possible for y to have copied a PARENT relationship from x, such that the pointed-to node is not a parent of y. If y is not a newly created node, then there are several potential violations of the invariants:

1. Invariant 1 may be violated if y is a TopLevelNode: the JOB relationship out of y leading to some RootProcessNode may have been deleted. Invariant 1A may also be violated, since there may be two arcs pointing to a root process node, each labeled with the same (JOB,name).

2. Invariant 2 may be violated if y is SystemProcessNode. Important secondary relationships may have been deleted.

3. Invariant 6 may be violated since the original PARENT out of y may have been deleted.

4.2 Copy_Tree Semantics

```
procedure Copy_Tree(FROM, TO_BASE: in Nodes;
(To_Rel,To_Key): Labels);


1. raise USE_ERROR => (exist z INSET Tree(FROM))(z INSET
   ProcessNodes
     UNION DeviceNodes);
    --if there are any process nodes or device nodes in
   the FROM tree,
    --raise USE_ERROR exception.
```

```
2.  (exist y INSET out Nodes)(y not INSET in Nodes and out
    P(TO_BASE, (To_Rel,To_Key),y) and out
    S(y,Parent,TO_BASE);
     --a new node, y, has been created, whose parent is
    TO_BASE. y will
     --be the root of the copied tree.
     --Let us refer to this y as NewRoot.

3.  (forall z INSET Tree(FROM))(forall seq INSET Labels*)
    (exist w INSET out Nodes)
     (w not INSET in Nodes and (in P*(FROM,seq,z) -> out
    P*(NewRoot,seq,w));
     --The basic tree structure has been copied; ie, every
    primary
     -- path emanating from FROM has a corresponding
    primary path
     --emanating from y, leading to a newly created node
    w.

4.  Definition:  (forall z INSET in Tree(FROM))(forall w
    INSET out Tree(NewRoot)), we define w to be
    CorrespondingNode(z) iff
     (forall seq INSET Labels*)(in P*(FROM,seq,z) -> out
    P*(y,seq,w)).  That is, w equals CorrespondingNode(z)
    iff w occupies the same relative position in the
    copied tree that z occupies in the original tree.

5.  (forall z1,z2 INSET Tree(FROM))(forall R INSET Labels)
     (in S(z1,R,z2) -> out
    S(CorrespondingNode(z1),R,CorrespondingNode(z2));
     --all secondary relationships strictly within the
    original tree have
     --been recreated in the copied tree.

6.  (forall z1 INSET Tree(FROM))(forall z2 not INSET
    Tree(FROM))
     (forall R INSET Labels)
     (in S(z1,R,z2) -> out S(CorrespondingNode(z1),R,z2));
     --all secondary relationships emanating from the
    original tree
     --to a node outside the original tree have been
    copied so they
     --also emanate from the corresponding node in the
    copied tree and
     --terminate at the same pointed-to node.

7.  (forall z1 INSET Tree(FROM))(forall z2 INSET out
    Nodes)
     (z2 = CorrespondingNode(z1) -> (out CONTENTS(z2) = in
    CONTENTS(z1)
     and OUT ATTRIBUTES(z2) = in ATTRIBUTES(z1)));
     --Nodes and attributes have been copied in addition
    to the
     --tree structure.
```

4.2.1 Analysis of Copy Tree The following invariants may be violated by the Copy_Tree semantics:

1. Invariant 7: This may be violated for NewRoot since the semantics call for the PARENT relationship to duplicate the PARENT relationship out of FROM. In addition, Invariant 11 may be violated since there may be two outgoing PARENT arcs emanating from y.

4.3 Semantics of RENAME

```
procedure RENAME(RenamedNode,NewParentNode: Nodes;
newRelation: Label);


   1. raise USE_ERROR => NewParentNode INSET in
      Tree(RenamedNode);
       --If the node which we would like to be the new
      parent is already
       --in the tree of the renamed node, then there would
      be a circularity
       --among primary arcs if we were to put a new primary
      arc from
       --NewParentNode to RenamedNode.  Thus an exception is
      raised.

   2. raise USE_ERROR => (exist x INSET in
      Nodes)(x/=RenamedNode and
       in P(NewParentNode,newRelation,x));
       --If the parent node already bears the desired
      relation name to
       --some other node, then raise USE_ERROR.  Otherewise,
      there would
       --be two primary arcs out of the parent node with the
      same label.

   3. out P(NewParentNode,newRelation,RenamedNode);
       --The desired primary relationship has been created.

   4. out S(RenamedNode,PARENT,NewParentNode);
       --The corresponding secondary PARENT relationship has
      been created.  (forall n INSET Nodes)(forall R INSET
      Labels)
       (n/=NewParentNode -> not out P(n,R,RenamedNode);
       --In the new state, there is no node other than
      NewParentNode which
       --bears a primary relationship to RenamedNode.
```

4.3.1 Analysis of RENAME The invariants appear to remain invariant under the semantics of RENAME. Note that if the renamed node is a process node, the UserAssociatedWith it may change. This might be an unintended side effect.

UNIX AS THE CAIS?

Richard Thall

SofTech, Inc.

In the course of creating both the Common APSE Interface Set (CAIS) and the Ada* Language System (ALS), the author has often been confronted with the question of why the UNIX** operating system interfaces are not adopted as the Common APSE Interface Set. It is also suggested that the UNIX tool set be used as a base for a standard MAPSE. Some UNIX tools, such as MAKE, have a particularly devoted following. The suggestion to use UNIX has some attractive benefits and some serious problems. This paper discusses some of the motivations and benefits for using UNIX and contrasts these with the potential problems.

The motivations behind the suggestion to adopt UNIX include:

- o the use of a proven, flexible operating system, vis-a-vis a non-existent and unproven CAIS,
- o the proven portability of UNIX vis-a-vis the questionable portability of the rather complex CAIS,
- o the popularity of UNIX would provide a large initial user base, easing the cost and trauma of introducing the CAIS,
- o the existence of a ready pool of knowlegeable support programmers,
- o the existence of a large library of tools providing a rich base for Ada programming environments,

Some favor the adoption of UNIX as a model or base for the CAIS. This would substantially simplify the problem of CAIS design and reduce the risks involved in implementation. The advantages of adopting UNIX are clearly substantial and difficult to dismiss. However, there are significant problems to be addressed on political, administrative, and technical fronts.

* Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office) OUSDRE (R&AT).
**UNIX is a trademark of AT&T Bell Laboratories.

The author believes that a UNIX-based "CAIS" already exists, the ALS. UNIX was used as a model for the design of the Army's Ada Language System. The adoption of a CAIS design based much more closely on the ALS is one way to capture more of the UNIX flavor. The design of the ALS started with UNIX as a model, simply because UNIX has a track record of proven portability; and portability is the first requirement of the STONEMAN. To UNIX were added capabilities that were necessary to satisfy the DoD requirements as set forth in the STONEMAN. These are:

a. attributes and associations for easier file structuring,

b. pathnames and a command language which are syntactically closer to Ada,

c. teams, user lists, append access, attribute change access, and via access for better access control needed to control program libraries,

d. revisions, variations, unique identifiers, sharing, and derivations for configuration management, and

e. task synchronous, process asynchronous input-output.

It would not be unreasonable to characterize the ALS as a form of UNIX which has been revised to satisfy the STONEMAN requirements.

Non-Technical Issues----------

The three most difficult problems with the adoption of UNIX are not technical problems at all. They are:

a. Who controls the definition of UNIX?

b. What is the definition of UNIX ?

c. Who will support UNIX ?

The first, and very likely the most troublesome, problem is a political one. UNIX is the property of AT&T, while the CAIS is a public effort being carried out by the U.S. DoD with representatives from industry and academia. Since AT&T owns UNIX, it effectively controls its definition. An entirely independent CAIS, on the other hand, would be controlled and developed by the DoD. Maintaining configuration control of a CAIS based upon UNIX, would require rather intimate coordination between AT&T and DoD. It is far from clear that either organization would be willing (or even should be willing) to submit to the constraints that would be necessary to establish a workable relationship.

The second problem is establishing a clear definition of UNIX. What is "standard" UNIX? Brian Kernighan, one of the originator- of UNIX was once asked a question like 'in standard UNIX what happens when ...'. His reply was "I've never seen an unhacked UNIX." Meaning that even he did not know what a "standard" UNIX is. UNIX is a "hackers" system. It invites programmers to "tweek" it. Almost every UNIX installed has some sort of modification that presents a potential pitfall to the cause of interoperability and transportability (I&T). AT&T itself has issued many versions of UNIX and there are endless UNIX look-alikes which sometimes differ in important details. Suppose, however, a standard could be agreed upon, who would control it? Dealing with AT&T could be prohibitively expensive for the Government and for users. It would also tend to introduce a substantial amount of technical inertia in the initial design. (Although the author believes that this is more likely to be beneficial than not).

UNIX source licenses typically cost about $50,000 on a per-CPU basis. Would each CAIS using a modified UNIX require a source license? Copies of Berkeley UNIX cost about $40,200; $40,000 to AT&T for a license and $200 to Berkeley to cover distribution costs. It is possible that the DoD could make some master licensing agreement with AT&T. However, AT&T would view this, and rightly so, as handing DoD a major market-share and would require concomitant compensation. Tens of millions of dollars would be about the expected order of magnitude for opening negotiations.

The third problem is to establish a clear supplier of support for the standard. When the Army was starting the ALS, they did make an explicit decision to use VMS instead of UNIX. The reason was that they would have to support UNIX themselves, but VMS is fully supported by Digital Equipment. The decision has proved to be a wise one. VMS is a very robust operating system. It has a wide number of services. It seldom crashes. In developing the ALS no VMS modifications were ever needed. However, SofTech did need to modify Version 6 UNIX in order to get a much simpler UNIX-based environment running properly. I am not suggesting that VMS be adopted as a standard, but only pointing out that one organization, the US Army, felt that the support issue was significant enough to influence the acquisition decision. Even the Army did not want to undertake the job of providing ongoing support for an operating system. The Army is not alone in this. In developing the CAIS, we can expect that changes to UNIX will be needed. Who will service the modified UNIX for Ada users? In the past, UNIX has been a do-it-yourself operating system. But this is clearly unacceptable to the Ada and DoD communities where standardization is to be scrupulously maintained. If we must use a modified form of UNIX, then have we not obviated the advantages of using UNIX in the first place?

A significant step toward a solution for these problems has been made by AT&T. In January of 1983, AT&T announced UNIX System V, promised to develop it in an upward-compatible and controlled way, began selling support for source licensees, and pledged to work with the user community to establish UNIX standards. Since that time, AT&T has worked with "/usr/group" to publish C library standards and command syntax standards, and is now working to publish system call standards. At the same time, AT&T signed porting agreements with several vendors and has developed a UNIX system certification test which defines an implementation as being in conformance with UNIX System V. At the January 1984 Usenix conference in Washington, D.C., Jack Scanlon, Vice President of the Processors and Systems Software Division of AT&T Technologies, pledged AT&T to remain committed to this definition of the base UNIX system.

From these activities, it is clear that a standard definition of UNIX is emerging and that AT&T is committed to its promotion and support. It is also clear that AT&T is presently interested in the porting of UNIX to any and all vendor hardware. However, is this the standard that will answer DoD requirements? What will DoD users pay for using this standard? What will they gain? How will DoD control the evolution of such a standard? Will AT&T's

recent entry into the hardware market eventually compromise the present commitment to UNIX portability?

Technical Issues----------

The major technical problem with UNIX is:

a. How is Ada tasking to be supported?

Other technical problems are:

b. Most UNIX implementations have a 14 character limit on node names.

c. UNIX traverses directory structures sequentially (slowly).

d. UNIX has no built-in support for configuration management (revisions, etc.).

e. UNIX has a rather limited access control model.

f. With minor exceptions, UNIX supports only a strict hierarchical file structure.

g. There are no definable node attributes in UNIX.

h. In spite of the recent wave of discovery, the UNIX design is well over a decade old.

The major technical drawback of UNIX has to do with support of Ada tasking on a computer running UNIX. An Ada program consists of a number of asynchronously executing tasks. When one task waits for completion of an input-output (IO) operation, the other tasks continue executing. Tasks are synchronized at discrete points with the rendezvous mechanism. Rendezvous are intended to occur very frequently, and they are intended to be serviced very quickly. Ada tasks can be allocated and deallocated very quickly. There can be an arbitrary number of them, hundreds, or even thousands.

A UNIX system executes a number of processes. Generally, the number of processes is limited by the size of a static table. UNIX systems are typically configured for 50 or 100 processes to service the users with a dozen or so processes dedicated to system chores. In "standard" UNIX, IO is process synchronous; in other words, when a process issues an IO request, the whole process waits until the request is satisfied.This is not quite as bad as it sounds. First, the process does not have to wait long for output requests to be satisfied. The process needs to wait only until the operating system has moved the output to an in-memory buffer. On input from a terminal, the operation can be bypassed if there is no input immediately available; in which case the read can be retried later. This prevents the process from becomming blocked when waiting for direct input from a user. However, for an input operation from a device other than a terminal, a UNIX process waits until the operation is physically completed. There are a number of privately modified UNIXs that do process asynchronous IO for real-time applications.

Before examining this problem in detail, it is important to draw the distinction between the roles of the Ada Programming Support Environment (APSE) host and the Ada target. UNIX was designed to serve as a rich software development host; a goal that most will agree has been well achieved. UNIX was never intended to be used as a high-performance real-time target. In serving as a base for a CAIS, or a replacement for a CAIS, it is entirely sufficient that UNIX fulfill the APSE host role. However, an APSE host is also an Ada target. Certain minimal performance must be provided and full Ada semantics must be achievable. It is not clear if a degenerate implementation of tasking is allowed under Ada semantics nor is it easy to discover if such an implementation is likely to have any unacceptable side effects.

In implementing Ada on UNIX, it will be necessary to, somehow, map Ada tasks onto UNIX processes. There are four choices:

a. use one UNIX process for each complete Ada program with all its tasks,

b. use one UNIX process for each Ada task,

c. use one UNIX process for each program with a number of IO server processes, or

d. use one UNIX process for each Ada program, but add asynchronous IO capabilities to UNIX.

The first alternative will yield a system where all tasks wait while IO is completing for any task. While this is definitely outside the spirit of Ada, we may find that this is acceptable for some classes of APSE hosts.

The second alternative has other problems. The number of tasks that could be allocated by all concurrently executing Ada programs would be limited by the size of the UNIX tables. Task allocation would be slow because UNIX process creation is slow; it was never intended for this type of use (or abuse). The author does not view this as a feasible alternative.

The third alternative, is a compromise. A predefined small number of server processes are allocated to do IO for the process executing the Ada program and all of its tasks. Or, alternatively, the Ada tasks are distributed among a predefined small number of processes. If there are N servers, then one achieves N-way concurrency. This may or may not be acceptable for a given application.

The author regards the last alternative as the most satisfactory, given the essential design constraints of UNIX. This has already been done in some privately supported versions of UNIX. There seem to be no serious technical barriers. While this will not result in a high-performance tasking implementation, it should satisfy Ada semantics well enough for host use. What results, however, does not strictly conform to the definition of UNIX. While "upward" compatibility is unaffected, the ability to move at least some Ada tools to "normal" UNIX implementations is sacrificed. Is this acceptable?

The remaining minor technical items (b-h) are simply a list of the more obvious limitations that would be visible to Ada users. If UNIX is adopted, then the Ada community by default, "buys" the entire list of arbitrary and not-so-arbitrary UNIX limitations, rules, syntax, semantics, and other aspects of UNIX culture. This also includes a ready-made list of bizarre tool names headed by cat and grep. Yes, most of these items are fairly minor and can be changed. Many of them have been changed in some versions of UNIX, e.g., Berkeley UNIX eliminates the 14 character limit on node names and also has a faster file system. (It is interesting to note that a large share of th

Berkeley effort was funded by DARPA and other DoD sources.) However, if extensive modifications are made for Ada, have we destroyed the benefits of adopting UNIX in the first place?

If adopted, UNIX interfaces that can be used by Ada tools will have to be designed and implemented. A difficult problem will be the management of Ada program libraries. This has proven to be an exceedingly complex problem which taxes the capabilities of many file systems. It is made more complex in UNIX by the absence of a revision mechanism in the file system and the rather primitive access control capabilities. The MAKE tool is often cited as one of the important benefits of adopting UNIX. However, the need for MAKE is eliminated if a proper Ada program library manager is implemented.

Conclusion----------

The advent of UNIX system V is a major step forward in providing a definition of UNIX suitable for a standardization effort. This largely resolves the definition problem and the support problem, but the problem of control and ownership of the standard remains an open question. We must also ask ourselves the basic question, 'does UNIX meet the DoD requirements?'

Acknowledgements----------

The author would like to thank J. Eli Lamb of AT&T Bell Laboratories who provided numerous suggestions for improving the technical accuracy and balance of this paper. The author would also like to thank Tricia Oberndorf for suggesting a number of clarifications.

Ada Configuration Management Workshop

TECHNICAL REPORT

Prepared by

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

for

Ada JOINT PROGRAM OFFICE
WASHINGTON, D.C. 20301

1 SECTION 1 - INTRODUCTION

The primary goals of management, when developing a hardware/software product, are to ensure that the product is delivered on time and within budget, and to ensure product integrity. Product integrity is defined to mean that the product meets or exceeds the requirements or expectations of the end user. The management guidelines for assuring product integrity for hardware products are better established than those for software products. Typically, when a hardware/software product is being developed, the techniques for ensuring product integrity of the hardware are applied in microscopic detail, while the software is viewed as a single unit to be delivered upon installation of the total product. The body of techniques applied to hardware development to ensure product integrity and timeliness cannot be applied to software development without substantial modification. Therefore, to aid management in meeting the above goals with respect to software, the following supporting disciplines are evolving:

1. Configuration Management (CM)
2. Verification and Validation (V&V)
3. Test and Evaluation (T&E)
4. Quality Assurance (QA).

Configuration Management is defined by the Electronic Industries Association to be:

o A discipline applying technical and administrative direction and surveillance to:

1. Identify and document the functional and physical characteristics of a configuration
2. Control changes to those characteristics
3. Record and report change processing and implementation status.

A configuration is an aggregation of software components and of the relationships among the software components. The need for software CM arose as the complexity of software increased. With this increase in complexity, software could no longer be viewed as a single deliverable product, with no inherent structure, no intermediate deliverables, and no updates after delivery.

Verification and Validation is the discipline that addresses the issues of software fulfilling performance requirements and of ensuring that specified requirements are stated and interpreted correctly. Verification is the activity carried out to ascertain whether or not a product satisfies specified requirements. Validation is a subjective activity that ensures the end product meets the user's needs.

Test and Evaluation is the discipline that is imposed on the creating organization by an outside organization. The outside organization executes pre-defined test procedures to assess whether or not the product meets the specified requirements.

Quality assurance as a software discipline has not been well defined or uniformly treated. In practice, there are four loosely defined QA models which are applied to the software life cycle.

1. In the first model, QA is a concept, rather than a specific activity, which begins on the first day of a project and continues throughout the life cycle of the project. In this model, quality is achieved through performance of the activities that comprise CM, V&V, and T&E.

2. In the second model, QA is a separate activity that is carried out after CM and consists of V&V and T&E.

3. In the third model, QA is a separate activity that is carried out after the development of a product and consists of CM, V&V, and T&E.

4. In the fourth model, QA is a specific activity that is carried out after CM but precedes V&V and T&E.

The existence of several models underlines the inherent difficulty of evaluating the quality of a software product.

Early in 1983, the Ada* Joint Program Office became concerned about these disciplines and their relationship to Ada and emerging Ada Programming Support Environments (APSEs). Since little formal work had been done to answer the concerns, it was decided to hold a two-day workshop on the first and most pressing of the disciplines, Configuration Management. The Naval Ocean Systems Center together with the Computer Sciences Corporation organized and hosted a Configuration Management Workshop (CMW) in San Diego, California, on 7-8 June 1983.

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

The attendees (Appendix A) at the CMW were chosen for their diversity of backgrounds in CM and/or Ada. Representing various companies and DoD organizations, they brought to the workshop considerable breadth of experience. They worked in four small groups whose topics were:

1. CM Definition and Aspects
2. What's Unique About CM for Ada
3. User's CM Needs
4. Tools for Ada CM.

Section 2 of this report contains a bibliography of documents relative to the CM discussions. Section 3 contains the analysis of the CM issues that were discussed during the Workshop. Section 4 presents a summary of the recommendations regarding CM relative to Ada drawn from the Workshop. Appendices A and B present the Workshop attendees and agenda respectively. Appendix C presents the workshop and full session reports. Appendix D presents a Configuration Management Overview distributed to the attendees.

2 SECTION 2 - BIBLIOGRAPHY

1. Ada Integrated Environment Draft System Specification, Prepared for Rome Air Development Center, Intermetrics, 15 March 1981.

2. Ada Integrated Environment Interim Technical Report, Prepared for Rome Air Development Center, Intermetrics, 15 March 1981.

3. Ada Language System Specification, U.S. Army CECOM, Ft. Monmouth, NJ, Contract No. DAAK80-80-C-0507, Draft Document CR-CP-0059-A00, June 1981.

4. Bersoff, E. H., V. D. Henderson, and S. G. Siegel, Software Configuration Management, An Investment in Product Integrity, New Jersey, Prentice-Hall, Inc., 1980.

5. Bryan, W., C. Chadbourne, and S. Siegel, Tutorial: Software Configuration Management, IEEE Computer Society Press, EHO 169-3, 1980.

6. Configuration Management System Program Performance Specification Report, Prepared for Naval Ocean Systems Center for Contract No. N00123-80-D-0364, Computer Sciences Corporation, 1 April 1983.

7. Larson, J. A., Tutorial: End User Facilities in the 1980's, IEEE Computer Society Press, EH0198-2, 1982.

8. Parikh, G. and N. Zvegintzov, Tutorial on Software Maintenance, IEEE Computer Society Press, EH0201-4, 1983.

9. Podell, H. J. and M. Weiss, Computers and Business, How Managers are Using Computers to Support Their Decision-Making, IEEE Computer Society Press, 1981.

10. Putnam, L. H., Tutorial, Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers, IEEE Computer Society Press, EH0165-1, 1980.

11. Reference Manual for the Ada Programming Language, U.S. Department of Defense, ANSI/MIL-STD-1815A.

12. SIMON, A Functional Description, Computer Sciences Corporation, 1983.

13. SIMON, User's Manual, Version 1.0, Draft Document, Computer Sciences Corporation, June 1983.

14. Feldman, S. I., Make - A Program for Maintaining Computer Programs, Bell Laboratories, 15 August 1978.

3 SECTION 3 - ANALYSIS OF CM ISSUES

This section contains the analysis of the CM issues that were discussed during the CMW and includes relevant terminology.

3.1 Software Problem Addressed by CM

Software components exist in two basic forms, that is, a non-derived form and a derived form. The non-derived form includes specifications (requirements and design), source code, testing procedures, and other supporting textual information. The derived forms (i.e., object code and executable code) are obtained from non-derived software components and other derived forms through a translation process, such as compilation. Configuration Management maintains such correspondence and identity among software components, regardless of their form.

On any project the following software activities begin almost simultaneously and continue in parallel:

1. Creation of the software
2. Modification of the software
3. Definition of a configuration of the software
4. Modification of the definition of a configuration of the software.

The absence of sequentiality in these activities can create chaos unless the discipline of CM is applied.

The Department of Defense (DoD) has recognized this software problem and that CM helps resolve it. Therefore, on 1 May 1979 the DoD issued directive number 5010.19 for CM on DoD contracts.

In summary, the DoD reinforced the concept that Configuration Management aids the software creator by providing a mechanism for:

1. Identifying a configuration
2. Controlling changes to a configuration
3. Performing status accounting on a configuration
4. Auditing a configuration.

Identification of a configuration is the means of describing the software components and the relationship existing among the software components relative to a specified baseline. A baseline is the content of a configuration at a designated and fixed time during the software life cycle. Software configuration control consists of applying procedures to document a change request, approve the change request, and track the implementation of the change to the baseline of the configuration. Status accounting is the mechanism for tracking back to the most immediate baseline of the configuration and reporting the results. Software configuration auditing is the mechanism for evaluating the status of the system relative to a pre-defined baseline. It is in software configuration auditing, as the disciplines are currently practiced, that software CM overlaps software V&V. Therefore, this area of CM received less emphasis at the CMW.

3.2 Software Configuration Identification

Software configuration identification consists of the following three steps:

1. Identification of the software components
2. Specification of the relationships among the software components
3. The definition of the baseline.

3.2.1 Identification

It is common practice to divide large software products into smaller, more manageable software components. As software components are created, the software creator and the host database management system assign a unique identifier to each component. The mere existence of these software components does not define a configuration that can be put under configuration management control or that can be used to define a baseline. Unique naming of software components is simply the first step necessary to start building a configuration.

3.2.2 Specification of Relationships

For a configuration or baseline to exist, the relationships among the components must be defined explicitly in the project database.

3.2.3 Defining a Baseline

After the relationships among the software components have been defined they can be uniquely identified, put under configuration management control, and then included in a baseline definition. The definition of the baseline is the final step in the CM identification process.

3.3 Software Configuration Control

Software configuration control consists of controlling those steps that must be performed in order to document a change request, approve the change request, and trace the implementation of the change to the configuration. In the software life cycle, there are three identifiable phases for which CM control is necessary:

1. Phase I begins when a software component is created by the programmer, documentor, or a test team member (i.e., software creator)

2. Phase II begins when the creator submits a configuration (consisting of one or more software components) to project-wide configuration management control. This configuration may or may not become part of a future baseline.

3. Phase III begins when the configuration becomes part of the baseline.

Each of the above phases is in itself an iterative process and the three as a whole form an additional iterative process. During Phase I, the software creator has access to the software creator's workspace, and read-only access to software components in Phase II or Phase III (i.e., software components that are already under project-wide configuration control). Further, the software creator does not have access to any other software creator's workspace. That is, the software creator may not build software components based on configurations that are not under project-wide configuration management control. Various systems support this concept although they may use different naming conventions such as "windowing" or "sharing".

When the software creator submits a configuration to Phase II, the configuration will be reviewed by a separate organization (possibly the QA, V&V, or T&E) for acceptance or rejection. If the configuration is accepted, it will be submitted to another organization (possibly T&E) for acceptance or rejection into the baseline. If the configuration is rejected at any approval point in Phase II, it will be sent back to the software creator for modification. The tracking of a rejection in Phase II may be informal, because the configuration was not put into the product baseline.

However, during Phase III, control is very formal, in that the software creator cannot change the product baseline without formal approval from an organization such as the Software Configuration Control Board (SCCB).

3.3.1 Phase I Control

As stated earlier, configuration control during Phase I is very informal and exists only to the extent that the software creator chooses to define it. Typically, control during this phase does not include written change requests, written approvals for changes, or traces from change requests to actual changes. Rather, this phase is characterized by rapid creation, modification, and debugging. As a result, it is very easy for a software creator to forget which software components are interrelated and the exact sequence of command language commands needed to update the configuration. For example, after a long edit session, the creator may easily lose track of which software components have been changed. However, the software creator should be prevented from building software components based on software components that are not under Phase II or Phase III control.

3.3.2 Phase II Control

Configuration control in Phase II begins when the software creator submits an identifiable configuration to another organization such as an internal QA, T&E, or V&V organization. Phase II includes all control activities that exist between the time that the software creator had unlimited change control (Phase I) and the time that the software creator has no change control (Phase III). The subjectivity with which QA, T&E, and V&V are applied to a given product causes variations in the way controls are passed among the phases. For example, during this phase, control may be passed verbally among reviewing organizations. Therefore, configuration control during Phase II is not as well defined as control in Phase I or Phase III.

3.3.3 Phase III Control

Configuration control during Phase III is characterized by a formal sequence of steps that must be followed to change a software component in the baseline:

1. A written change request is submitted to the SCCB after a bug

has been discovered or when a new feature is desired

2. The SCCB evaluates and approves or disapproves the change request

3. If a change request is approved, the SCCB assigns the change request to a software creator who determines which software components would be affected and modifies them to reflect the change request

4. The software creator submits the modified software components for approval, testing , and integration into the baseline.

It was the consensus of the CMW that the above steps should be automated. In addition, it was noted that Step 2 contains subjective aspects that cannot be fully automated. Further, it was recognized that the automation of these steps is within the state-of-the-art.

3.3.4 Summary of Software Configuration Control

In summary, configuration control increases in intensity from Phase I to Phase III. During Phase I, the software creator has unlimited freedom to make changes to his configuration. During Phase II, there is a wide variation in the application of controls. However, during Phase III, it is generally agreed that configuration controls must be strictly enforced. At the CMW, it was stressed that the controls should be carefully balanced to achieve the following two goals:

1. Protection of the baseline from unauthorized modification

2. Support of the creator's productivity.

3.4 Software Status Accounting and Reporting

Software configuration status accounting is the mechanism by which the outputs of the other three CM activities (identification, controlling, and auditing) are:

1. Recorded and stored in the configuration status accounting database

2. Reported upon from the above database.

The information stored in the configuration status accounting database provides the basis for tracking the evolution of the project-wide configuration from a given baseline. The information can be used to determine such things as productivity, cost to complete the product, and impact of rescheduling and reassignment of tasks and personnel. Also, status accounting can be used to track changes from the established baseline to the current state of the configuration. The information derived from this tracking can be used to determine whether or not to establish a new baseline.

Software configuration status accounting should be automated because of the large amount of information recorded and the variety of reports desired. Automation of these time-consuming and tedious tasks will increase developer and manager productivity.

4 SUMMARY

This section summarizes recommendations, regarding CM relative to Ada and APSEs, drawn from the CMW and from current literature. Included in this summary is a proposal for future CM workshops.

4.1 Recommendations

All CM activities should be automated to the extent that is possible within the state-of-the-art. CM should not be a burden or extra activity for the user. That is, as much CM as possible should be transparent to the user. The parts of CM that are not transparent to the user should aid and not hinder the user

It is the responsibility of the CM control activity to maintain control over project-wide configurations (i.e., over both the software components comprising the configuration and the relationships among the software components). This activity should control neither the contents of the software components (this is a QA function) nor the definition of the relationships among the software components (this is a management/technical function). Rather, the CM control function should have the responsibility for maintaining control over changes to the baseline of the configuration within a control phase and across the control phases. Moreover, this activity consists mainly of bookkeeping.

It is important that CM never be an afterthought to a project but rather that CM be an integral part of the project from the beginning. Relative to APSEs, there is the opportunity to include within the APSE a standard database interface and an integrated toolset that allows CM to be an integral part of any project.

Since the APSE database interface will be standardized, it is important that this interface include:

1. The capability for defining a data dictionary (that is, for defining the structure and contents of a database) that supports CM
2. The functional interfaces for easily implementing the CM activities discussed in this report.

In addition, it is recommended that the following tools be included in an integrated CM toolset:

1. A "build" tool that provides a means for the creator to define and build a view of the project-wide software configuration for modification
2. Automated "Configuration Object" builder tools (e.g., one

such tool will build a "Configuration Object" for the compilation and linking order for a software configuration)

3. A "Configuration Object" library manager tool that provides functional support for Ada program libraries in APSEs.

The CMW concluded that APSEs do not present new CM problems for development projects. However, emphasis should be placed on this unique opportunity to build an advanced CM system that could be made available to every DoD contractor. Frequently, time is wasted on contracts while defining and building a CM system. All of that CM effort is usually out of the mainstream of the DoD contract and success with the CM system is usually not realized. APSEs with appropriate CM systems would remedy this situation.

The CMW also concluded that the Ada language does not present new CM problems. The packaging concept and the separate compilation facilities provided by the Ada language definition can be aids to CM. The Ada language, however, does not preclude the need for CM (See Working Group 2 Report, Appendix C).

4.2 Future CM Workshop

It was abundantly clear that the two-day CMW was not sufficient to adequately uncover or address all the CM issues. It was agreed, however, that neither the APSE nor the Ada language present new CM problems, but rather present opportunities to improve CM and the other product integrity disciplines (QA, T&E, and V&V). It is recommended that future CM workshops be held and that they continue to aim discussions explicitly at improving CM in the APSEs. (Prior to the workshop, position papers should be solicited to limit attendance to those individuals who have personally contributed to CM on previous projects.) It is also recommended that the following issues be among those addressed at future workshops:

1. The database interface to the APSE should be standardized. Configuration Management issues should be given special consideration during this standardization process.

2. The amount of human control needed in an APSE CM system should be investigated. The CM process should not be manual, but it should be identified where it is absolutely necessary for a human to authenticate the contents of the configuration management database.

3. It should be determined how CM overlaps with the other product integrity disciplines (QA, T&E, and V&V) and how all of the product integrity disciplines, not just CM, fit into an APSE

4. It should be determined how much and what kind of information should be gathered for what kind of reports for CM on an APSE.

It is also suggested that the name of the workshop eventually be changed from CMW to Product Integrity Workshop for the DoD. This name change would broaden the scope and would emphasize the real problem to be solved.

APPENDIX A

ATTENDEES

NAME AND ADDRESS	PHONE NO.
Ramesh Babu ROLM MSC Mail Stop 150 One River Oaks Place San Jose, CA 95134	(408) 942-7703 (Work) (408) 554-9972 (Res.)
Mitch Bassman CSC 6565 Arlington Blvd., MC 281 Falls Church, VA 22046	(703) 237-2000 ext. 7038
Rebecca Bowerman CSC 6565 Arlington Blvd., MC 281 Falls Church, VA 22046	(703) 237-2000 ext. 6639
Jack Foidl TRW DSG 3420 Kenyon St., #202 San Diego, CA 92110	(619) 225-9400
Bob Fritz SAI-Comsystems 2801 Camino Del Rio San Diego, CA	(619) 293-7500 (SAI) (619) 225-6515 (NOSC) (619) 485-7702 (Home)
Alan Goldfine National Bureau of Standards Technology A-265 Washington, D.C. 20234	(301) 921-3491
Leah Hammond IBM Corp. Owego, NY 13827	(607) 751-2023
Larry Johnston NADC, Code 503 Warminster, PA 18974	(215) 441-3145

Bill Jordan/Lloyd Stiles (619) 225-2569
FCDSSA-SD AV 933-2569
200 Catalina Blvd.
San Diego, CA 92147

Liz Kean (315) 330-4325
RADC/COES AV 587-4325
Griffiss AFB, NY 13441

Henry Lefkovits (617) 456-3517
Alpha Omega Group, Inc.
P.O. Drawer M
Harvard, MA 01451

Gil Myers (619) 225-7401
NOSC, Code 8322
San Diego, CA 92152

Tricia Oberndorf (619) 225-7401
NOSC, Code 8322
San Diego, CA 92152

Alan Olson (619) 225-8401
CSC ext. 251
4045 Hancock St.
San Diego, CA 92110

Brian Ropside (214) 462-5756
TI
P.O. Box 405, MS 3407
Lewisville, TX 75056

Brian Schaar (202) 694-0212
Ada Joint Program Office
Room 3D139 (400 AN)
The Pentagon
Washington, D.C. 20301

Oscar Shapiro (617) 271-2474
Mitre Corp.
Box 208
Bedford, MA

Tom Smith (703) 237-2000
CSC ext. 516
6565 Arlington Blvd., MC 281
Falls Church, VA 22046

Dave Vatsaas (612) 456-2893
Sperry Corp.
Computer Systems
Sperry Park
P.O. Box 43525, MS U2T18
St. Paul, MN 55164-0525

APPENDIX B

AGENDA

CONFIGURATION MANAGEMENT WORKSHOP

A G E N D A

Tuesday, June 7, 1983

0830 - 0900 Arrive

Coffee

0900 - 0930 Welcome and introductory remarks

- Tricia Oberndorf, Chairman
- CSC hosts

Introductions

- Each attendee is asked to give their name, organization background/project that brings them to this workshop, and a one or two sentence position statement on some CM issue which should be considered at this workshop.

0930 - 1030 CM Definition (Group discussion)

1030 - 1100 Break

1100 - 1230 CM Experiences (15 minutes each) (prearranged speakers)

- Air Force
- Army
- Navy
- NASA

- AIE
- ALS

1230 - 1400 Lunch

1400 - 1430 Industry perspective (prearranged speakers)

1430 - 1500 Organize into the following working groups:

- CM Definition and Aspects
- What's unique about CM for Ada?
- User's CM Needs
- Tools for Ada CM

1500 - 1530 Break

1530 - 1700 Working Groups meet

Wednesday, June 8, 1983

0830 - 0900 Arrive

Coffee

0900 - 1030 Working group reports (summarize prior day's efforts)

1030 - 1100 Break

1100 - 1230 Working Groups meet

1230 - 1400 Lunch

1400 - 1500 Working Groups meet

1500 - 1530 Break

1530 - 1700 Working groups reports, discussion, conclusions, and closing

CONFIGURATION MANAGEMENT WORKSHOP

AREAS FOR DISCUSSION

June 7-8, 1983

1. What is configuration management in general? We need a solid definition from which to start. Questions include whether we are limiting ourselves to the code or if configuration management includes design and other phases and if it includes other documents besides modules of code. The kinds of issues we need to resolve are if configuration management should be restricted to the management of code development and release, or if it includes the management of documents related to requirements and design.

2. What new challenges/requirements does Ada impose on configuration management?

3. What new opportunities does Ada provide for configuration management?

4. What are the aspects of configuration management? One thinks of configuration identification, configuration tracking and reporting, and configuration changing; there are probably others.

5. What role does the program library and its manager play in configuration management?

6. What are the project manager's configuration management needs?

7. What are the programmer's (i.e., implementor's) configuration management needs?

8. What are the user's configuration management needs?

9. What good leads to configuration management exist (e.g., successful project experience, standard documents)?

10. What contributions do the AIE and ALS make?

11. What experiences have the AIE and ALS implementors had?

12. What experiences have other Ada implementors had?

CM EXPERIENCE - MILITARY AND INDUSTRY

From 11:00 a.m. to 12:30 p.m. and from 2:00 p.m. to 2:30 p.m., each person will be given the opportunity to give a five to ten minute informal talk about one or more of the following issues:

1. A configuration management plan you have written or implemented
2. A system you have managed or implemented or used that was under a successful or unsuccessful configuration control plan
3. Any topic on the "Configuration Management Workshop, Areas for Discussion" sheet.

APPENDIX C

WORKSHOP AND FULL SESSION REPORTS

Working Group 1 - CM Definition and Aspects

Chaired by: Alan Olson, CSC

Participants: Jack Foidl, TRW
Leah Hammond, IBM
Oscar Shapiro, Mitre
:loyd Stiles, FCDSSA-SD

Introduction:

The purpose of this working group was to define configuration management and discuss the implication of CM on our working environments.

Issues:

The following issues were discussed:

1. The state of the art for software engineering has come a long way, but Working Group 1 concurred that millions of lines of existing code do not perform acceptably. The procedures for incorporating the state-of-the-art software technology into the ALS/N with a high degree of enforcement is critical to a high quality, cost effective system. CM plays a key role in accomplishing this objective.

2. It is important to have technical expertise in the Software Configuration Control Board.

3. The Navy places heavy emphasis on software and hardware standardization, both near and long term. CM plays a key role in achieving and maintaining standardization with respect to documentation and coding standards.

4. The frequency and content of revision releases are the prime ingredients that determine how readily user sites will employ

a new revision and how effective it will be. CM has major control over these factors and should derive the most cost effective revision cycle for all products, including user applications.

5. CM should be automated as much as possible. The very act of doing anything should invoke the appropriate CM procedures automatically.

6. The CM system should be simple, flexible, and the effort to implement a change should not be so cumbersome that it stifles programmer (worker) productivity.

7. The CM database should contain data about the product line, the software components' derivation histories, and status accounting information.

8. In general, the CM discipline is well defined and widely employed via many different techniques and procedures. There are six different situations in which CM will be applied with respect to Ada:

    1. CM for Ada support software development

    2. CM for APSE development

    3. *CM for Ada applications (i.e., projects coded in Ada)*

    4. CM for Ada support software life-cycle maintenance

    5. CM for APSE life-cycle maintenance

    6. CM for Ada applications life-cycle maintenance.

9. All the standard ingredients of CM apply to each of these categories. However, a specific CM methodology needs to be defined and built into the APSEs. A specific CM methodology could be built into any programming environment, but the APSEs provide an opportunity to build an ideal CM system from the beginning.

Working Group 2 - What's unique about CM for Ada?

Chaired by: Mitch Bassman, CSC

Participants: Bob Fritz, SAI
Alan Goldfine, NBS
Gil Myers, NOSC

Introduction:

This working group was directed to discuss how features unique to Ada affect the requirements for configuration management. Participants addressed the Ada Language, Ada Programming Support Environments, and Ada design methodologies.

Issues:

The following issues were discussed:

1. The fact that a programmer is required to apply a WITH clause to a compilation unit to define dependencies upon other library units may facilitate some configuration management functions. Also, the smarter compilers and linkers, required to guarantee adherence to the rules of compilation order, will help CM.

2. An increase in the development and use of reusable software increases the importance of CM.

3. Generics will introduce some complications.

4. I/O issues require no special consideration. I/O falls under generics.

5. CM is needed to keep track of which pragmas, attributes, and pre-defined exceptions are supported by a given compiler.

6. In a mixed language environment (e.g., ALS/N) it is necessary to maintain a record of which language is being used in each source object under configuration control.

7. The existence of a standard programming language and standard interfaces will facilitate the development of a standard set of integrated CM tools.

8. CM is needed to keep track of what features are available in a run-time environment.

9. Using an Ada-based program design language (PDL) will allow consistent automated CM techniques to be applied throughout the lifecycle and will promote traceability of the design throughout the lifecycle as the system written in an Ada-based PDL evolves into code.

10. Systems to be coded in Ada are highly likely to be designed using an Ada-based PDL; this is in contrast to systems implemented in other languages, which rarely are designed using a PDL based specifically on the implementation language.

11. We must be careful not to blur the boundaries between PDL and code when using an Ada-based PDL. CM requires boundaries.

12. The package concept available in an Ada-based PDL facilitates the representation of data flow.

13. There is no significant difference between Ada and other languages for CM. The differences are all matters of detail.

14. Ada is an evolutionary language, not a revolutionary one; hence, changes in CM based on Ada will be evolutionary, not revolutionary.

15. Ada is not a miracle cure for solving the problems of CM.

16. Should automated CM tools in an APSE be visible or invisible to the user? That is, should a programmer (for example) have to invoke CM tools explicitly or should the underlying KAPSE (MAPSE?) manage his configurations for him? If both, what CM functionality should be applied implicitly and what must be explicitly requested?

17. It must be possible to determine the relationships between source file objects and compilation units.

18. If a project imports a package from a public library and depends upon a specific revision, then it is the responsibility of the project to configuration manage the use of the specific revision. If a package in a public library is updated with a new revision, the CM of the public library must guarantee that an earlier revision remains in the library as long as required by any user.

Working Group 3 - User's CM Needs

Chaired by: Rebecca Bowerman, CSC

Participants: Liz Kean, RADC
Brian Schaar, AJPO

Introduction:

The purpose of this group was to identify the users of a configuration management system and to identify their configuration management needs. In pursuing this goal, we avoided the use of industry or military organizational charts and names. We attempted to use descriptive words or phrases for the users and their needs.

Issues:

The following issues were discussed:

1. As a group, we support the ideas of fully automating the configuration process, and to achieve that, we acknowledge that the human interfaces to the system must be simple and "user friendly".

2. In all of our discussions, we found overlapping needs between configuration management and quality assurance and also between quality assurance and testing, all of which insure the integrity of a product.

3. We agreed that the goal of everyone involved in a product was to produce or to receive a high quality product. We also agreed on the description of many of the activities that must be performed to produce or to receive a high quality product.

4. We had some difficulty with definitions and semantics. For example:

    1. How do you assess the quality of a product?

    2. Is QA the discipline that includes CM, V&V, and T&E?

    3. Is CM just a bookkeeping activity?

    4. Where does CM stop and QA begin?

5. For the sake of expediency, we adopted the following philosophies/definitions:

    1. Quality assurance is a "state of mind" that exists throughout the life of a project

2. Quality assurance is also a separate activity consisting of Test and Evaluation (T&E) and Verification and Validation (V&V)

3. T&E includes the execution of a set of predefined test procedures. We avoided the issues of Independent Path Testing and Complete Path Testing because of limited time.

4. V&V is a subjective activity that includes reviewing test results and making judgement calls relative to specified requirements. We also did not spend much time on the details of this activity.

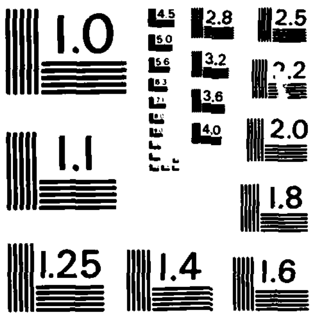
5. CM is all activities required to maintain consistency among related software components and all activities required to track changes to a product baseline

6. CM activities are more easily defined and automated than QA activities. However, there are many automated tools that can provide critical input to QA.

6. Figure 1 is an illustration of the steps that we agreed were necessary for ensuring product integrity. The boxes represent human interfaces and functions that may be performed by any organization. The functions may also be divided among organizations, both internally and externally. We made every attempt to avoid military and industrial organizational charts.

AD-A147 648 KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) 5/5
INTERFACE TEAM PUBLIC REPORT VOLUME 4(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P OBERNDORF 30 APR 84
UNCLASSIFIED NOSC/TD-552-VOL-4 F/G 9/2 NL

END
FILMED
DTIC


MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

The START box represents all discussions that take place before a user with a specific need is identified. The broad line connecting the NEED to the USER OF PRODUCT emphasizes the inseparability of the two. But it may also be the case that the NEED is identified at a higher level than USER. The first step towards getting a product to satisfy a need is to express the need to a PRODUCT MANAGER. The lines between the USER OF PRODUCT and PRODUCT MANAGER represent open communication between them throughout the life of a product. After receiving the description of the need, the PRODUCT MANAGER assigns the development of the product to a MANAGER OF PRODUCT CREATORS. The term CREATOR is used to emphasize that software is not just code.

As soon as the CREATORS begin developing the product, the activities that contribute to product integrity should already be in place. The consensus of Working Group 3 (WG3) was that, in this stage of product development, the activities that contribute to product integrity should be viewed as aids to the CREATOR and not restrictions on the CREATOR. For example, when the CREATOR is creating and modifying many software components, it is very likely that the CREATOR will not correctly recall all the command language commands necessary to update the CREATOR's configuration. We also discussed the automatic creation of a "description file" or "configuration file" for the CREATOR. These are the files that contain the definition of the dependencies among the software components in the CREATOR's configuration.

The timing for releasing partial development of a product from CREATORS was not part of WG3's discussion, but rather how and to whom the release should occur. The status of a partial release should occur with the help of an automated tool, to move the partial release from the CREATOR's workspace to a QUALITY ASSURANCE organization for review.

The details internal to the QA box were not explicitly agreed upon. However, we did agree that if QA rejects a configuration, the configuration will be returned to the assigned CREATOR. There should be an automated tool for rejecting a configuration and for accumulating reporting data for the CREATOR and the MANAGER of the CREATORS. If QA accepts a configuration, QA integrates it into the baseline of the product and submits the new baseline to the PRODUCT MANAGER for final approval and release.

The final stamp of approval before release should be performed by the PRODUCT RELEASE CONTROL STAFF within the PRODUCT MANAGER's box. Although they are routine and non-technical jobs, the jobs of packaging, of confirming that N copies of each deliverable item are included, and of updating final bookkeeping details were acknowledged as critical to ensuring the integrity of a product.

After the final stamp of approval, the PRODUCT RELEASE CONTROL STAFF released the product to the MANAGER OF PRODUCT MAINTAINERS. The PRODUCT MAINTAINERS have special needs for tools or documentation to help with learning the details of a product.

These needs seemed to be training issues which were not within the scope of WG3's purpose. However, once the MAINTAINERS are beyond the training stage, all of their CM needs were virtually identical to those of the CREATORS.

At any time in the life cycle of a product, anyone associated with the product may want to request a change to the baseline. Requests can come from USER OF PRODUCT, CREATOR, QA, PRODUCT MAINTAINER, or from inside the PRODUCT MANAGER's organization. There should be one central body for approving or disapproving changes to the baseline, namely, the Software Configuration Control Board (SCCB). Although the SCCB is a human function, there are many reports on the product, that can be generated from the product database, that can provide valuable information to the SCCB.

In summary, it was determined that CM is one of several disciplines that help insure product integrity and that CM consists of two functions:

1. Tracking, controlling, and reporting communications between the boxes, to maintain CM over the product as a whole

2. Maintaining consistency among related software components, as an aid especially within the CREATOR's box and within the MAINTAINERS' box.

Working Group 4 - Tools for Ada Configuration Management

Chaired by: Thomas Smith, CSC

Participants: Ramesh Babu, ROLM
Larry Johnston, NADC
Henry Lefkovits, Alpha Omega
Brian Rospide, TI
Dave Vatsaas, Sperry

Introduction:

Tools for Ada Configuration Management (CM) included the following areas:

1. procedures and methodologies to perform CM

2. automated software tools to support CM throughout the entire software life-cycle.

Issues:

The following issues were discussed:

1. CM is a "human activity" that is assisted by the computer. ("Human activity" here implies that the authentication of software must be done by humans with the assistance of automated software tools.)

2. Any kind of CM system imposes an "implicit methodology" for software development.

3. Any "good" software methodology must include a CM methodology to obtain a coherent approach to useable, reliable, and maintainable software that satisfies requirements

4. CM should never be an "after thought" for a project.

5. For automated CM software tools the "user" interface is of paramount importance.

6. Automated CM software tools must be "integrated" and "transparent". "Integrated" here implies that for any configuration management system within an APSE the many functionalities of the system must work in harmony with each other within the environment; "transparent" here implies that the program developer (whose software and/or documentation is being monitored or controlled) would not have to be actively involved in gathering data for CM.

7. What is a common "minimal" set of automated software tools for CM?

8. What classifies as a software "tool"?

9. A problem of automated software tools is the complexity of these tools and the requirements that these tools impose on the user (e.g., tools that perform requirement traceability throughout all stages of the software life-cycle such as PSL/PSA and SREM).

10. How does one obtain a workable mechanism for obtaining a "software baseline" using Ada program libraries? The "baseline" must not be corrupted while the program developers have accesss to it for the purpose of "unit testing" and "integration testing".

11. How are Ada program libraries to be recompiled to maintain consistent "baselines"?

12. With the advent of "off the shelf" and "reuseable" software, how will a consistent "software baseline" be maintained?

13. The previous three issues also address the issue of the migration of source and/or object files between Ada program libraries.

14. How will CM handle both the software environment database and the Ada program library database?

15. Standardization of the interface to the database(s) in an APSE is of primary importance in addressing the other issues brought up by this working group.

16. The database interface in an APSE must address Ada program library management and database requirements (i.e., access control, integrity, versioning, baselines, etc.) relevant to CM.

17. How does one maintain consistency between a central CM library and working software development libraries?

18. A CM database model, encompassing both developer needs and maintainer needs, must be specified.

19. A model for a standard CM database interface specification for an APSE must include entities, attribute, and relationships relevant to CM. (A proposed model discussed at length was the entity-relationship database model.)

20. A CM database in an APSE must be extensible (i.e., ability to add new functionality) and modifiable (i.e., ability to change functionality already present).

21. Automated CM software tools must be transportable between APSEs, and CM databases must be interoperable between APSEs.

22. Natural languages must be employed as much as possible in the

"user" interface for automated CM software tools.

23. An automated CM software tool must have a well defined function that addresses a large class of CM applications.

24. Automated CM software tools for status accounting, query language interfaces, report generation, control, costing, configuration identification, naming conventions, test data generation (including testable object generation), etc. must be developed for the APSE.

APPENDIX D

Configuration Management Overview

Configuration Management

An Overview

Prepared for

Ada Configuration Management Workshop
7-8 June 1983
San Diego, California

TRW

Defense Systems Group
San Diego California

Although configuration management is an esoteric term, its concepts and practices are universally applied by industry to deliver a product that meets customer requirements. Of course, the degree and consistency of application vary widely, depending on the complexity and use of the product. For sophisticated products built to stringent standards such as those imposed by the government for aerospace and military systems, the formalized discipline of providing uniform product descriptions, status records, reports, and change control is called Configuration Management. The formal implementation of Configuration Management is a major task involving numerous specialized concepts, personnel, and procedures.

A clear understanding of the concepts and techniques embodied by the terms "Configuration Management", "control", "identification", and "accounting" is essential to a successful Configuration Management program.

Configuration Management is the art of organizing and controlling planning, design, development, and hardware operations by means of uniform configuration control, identification, and accounting of a product. The goal of these objectives, is to assure that the delivered configuration items meet form, fit, and functional requirements. "Configuration" refers to a complete description of the physical and functional characteristics of a product; for example, its shape, size, materials, processes, power consumption, and performance (measurement range, accuracy, stability, linearity, etc.). *The term configuration* also applies to technical descriptions required to build, test, operate, and repair a configuration item.

The overall objective of configuration management is to guarantee the buyer that a given product is what it was intended to be, functionally and physically, as defined by contractural drawings and specifications and to identify the configuration to the lowest level of assembly required to assure repeatable performance, quality, and reliability in future products of the same type. To satisfy this objective the following five major goals are commonly an integral part of the configuration management effort:

a. Definition of all documentations required for product design, fabrication, and test.

b. Correct and complete descriptions of the approved configuration.
(Descritpions included drawings, parts lists, *specifications*, test procedures, and operating manuals.)

c. Traceability of the resultant product and its parts to their descriptions.

d. Accurate and complete identification of each material, part, subassembly, and assembly that goes into the product.

e. Accurate and complete pre-evaluation control and accounting of all

changes to product descriptions and to the product itself.

Achievement of the above goals will ensure a successful configuration management program. The quantity of data that must be identified, controlled, and accounted for, however, can present a major challenge to even the most talented administrators and managers.

Configuration Management applies technical and administrative direction and surveillance to configuration baselines and increased control as each baseline is established.

Baselining is the application of technical and administrative direction to designate the documentation which identifies and establishes the initial configuration identification of a product and that products configuration at specific times during its life cycle.

The basic techniques, methods, or procedures that enable the discipline of configuration management to be systematically applied are configuration identification, configuration control, configuration verification, configuration accounting and reporting, and configuration audits and reviews.

Configuration Identification refers to the technial documentation that identifies and describes the approved product configuration throughout the design, development, test, and production tasks.

Configuration identification consists of creating and formally releasing the complete technical documentation, including specifications, drawings, and data lists, which defines the original approved configuration (baseline) and subsequently defines all approved changes to the individual configuration items. It shall also be understood that this involves physical identification of parts, subassemblies, assemblies, and facilities as specified in drawings and specifications. It also involves the identifications of the design and as-built documents themselves so that they may be readily associated with the configuration identification they support. Configuration identification assures that hardware and all supporting documentation are continually compatible for the life of the configuration identification. It also applies to the identification of changes and to product markings. Identification, therefore, requires systematic control of part, specification, and data list numbers as well as the assignment of serial numbers. (Serial numbers enable identification of individual items that are otherwise identical).

Configuration control is a continuing function beginning in the earliest stages of a product and extending over the full service life of the individual configuration item. It consists of those systematic procedures by which configuration changes are proposed, evaluated, coordinated, and either approved for incorporation or disapproved. Initial plans of configuration control, before buyer acceptance of the first configuration item, are directed toward control of configuration as defined in documents, primarily specifications. Following first configuration item acceptance, configuration control is focused promarily on incorporation of changes into hardware and software documentation occurring as a result of approved specification and hardware changes.

Configuration Control involves the systematic evaluation, coordination, and approval or disapproval of proposed changes to the design and condtruction of a configuration item where configuration has been formally approved internally by the company or externally by the buyer, or both.

Having identified the approved product configuration and having controlled changes to the configuration up through the point of authorized release, it is then necessary to verify the physical incorporation of changes. Configuration verification involves continuous comparisons of the changed hardware or software with the engineering data/specification that defines its approved configuration. Inherent in this verification process is the discovery and correction of configuration discrepancies by (a) comparison of the configuration item to its approved configuration identification or (b) approval by the buyer of the configuration item's as built configuration, using officially prescribed procedures and forms to record the approval. The end product of configuration verification is a configuration item that is perfectly mated to all supporting technial documentation, or, failing this, that has all configuration discrepancies identified and specifically approved by the buyer.

Having accomplished the functions of configuration identification, control, and verification, we then make the previous tasks meaningful by the processes of configuration accounting for internal purposes and of configuration reporting for external (buyer) purposes. Configuration accounting is the recording and reporting of configuration item descriptions and all departures planned or made from the baseline through the comparison of authorized design data and the fabricated and tested configuration of the configuration item. This involves the creation of documentation, in formats specified by the contract, which is used internally and then reported to the buyer to provide him with accurate information on the configuration status of all configuration items entering his inventory or stock area. This reporting activity continues during the operational life of the configuration item, continuously reflecting changes authorized for factory or field incorporation.

In summary, the configuration status accounting technique establishes records which enable proper logistics support to be established. These records include (a) where a product is located or installed; (b) the identifications of selected product items by serial number makeup, and (c) current modification status. The complexities of the records must be consistent with the configuration identification and must be established by the buyer on a case-by-case basis to suit the required level of control and intended use.

Formal configuration reviews and audits are conducted to evaluate the progress of a configuration item. This process evolves through its conception, design, development, production, and test. These reviews and audits ensure that the physical and functional characteristics of a configuration item match those specified in the products specification. The reviews and audits to be conducted are as follows:

a. System Requirements Review

b. System Design Review

c. Preliminary Design Review

d. Critical Design Review

e. Functional Configuration Audit

f. Physical Configuration Audit

g. Formal Qualification Review

As in any field of administration and management, certain characteristics or qualities are particularly important to successfule configuration management. The key feature that characterize successful configuration control, identification, and accounting are as follows:

a. Early and complete definition of configuration management, goals,
scope, and procedures.

b. Speed in evaluating and processing changes.

c. Accurate identification and accounting of change.

d. Complete description of changes.

e. Close coordination among key elements of the project team.

f. Cooperative and responsive buyer.

g. Minimum labor requirements.

Two additional features are: (a) development of the simplest configuration control, identification, and accounting approach or system that will provide the desired results and (b) minimum number of forms and related documents for implementing changes and for providing complete records of all changes.

PROPOSED CHARTER

Ada RUN-TIME ENVIRONMENT WORKING GROUP (ARTEWG)

To establish conventions, guidelines, and performance criteria for the Ada Run-Time Environment that will facilitate the transportability of Ada application programs for production (embedded) systems.

ARTEWG AREAS OF INVESTIGATION

- o Elaboration of RTE dependencies for Appendix F of LRM;
- o Identification of other implementation dependencies (i.e., implementation required functions) inherent in RTE;
- o Delineation of RTE characteristics to specify manner in which dependencies may be satisfied; and
- o Investigation of interface requirements for expressing characteristics of RTE in source text of application programs and the code that implements the RTE.

!GENERAL COMMENTS FROM ADA-EUROPE AIE REVIEW

In April I sent you a contribution to the AIE review from the Ada-Europe Environment Working Group. Subsequently, we received two of the remaining B5 documents, PIF and DBUG, and reviewed these at our meeting on May 26th. At that meeting Knut Ripken reported on a trip to the USA. The impression he had gained, to quote the minutes, was that "AJPO want to accelerate ALS, slow down on AIE, and concentrate standardisation on ALS". Again to quote the minutes, "There was a general (technical) horror at the prospect of ALS as a standard. It was felt that in making detailed criticisms of AIE we had omitted to praise the overall design sufficiently". Most members had been involved in reviewing ALS previously so were aware of the differences. We would regret any slowing of the AIE design, as it incorporates some advanced concepts which would be well worth trying out.

!SPECIFICATION TITLE:	SYSTEM SPECIFICATION
!SPECIFICATION DATE:	12 NOVEMBER 1982
!SPECIFICATION NUMBER:	IR-676-2
!REVIEW DATE:	15 APRIL 1983

Paragraph	Page	Competence Level
# 0.0	1	C

It is regrettable that the AIE contains no proposals for tools to aid systematic testing of programs. Debugging is not testing and may be considered a less important part of software development.

It would be useful to be able to call MAPSE tools during a debugging session in the same way as this is permitted within an editing session. The current proposal of providing only a limited set of commands is a weak special case of this proposal.

The specification contains many new terms unfamiliar to most readers. It is suggested that the specification could be improved with the addition of a glossary.

It is our opinion that while the overall design may be technically adequate, the AIE still ignores the needs of the embedded computer applications. There is no AIE tools (i.e., compiler, linker, assembler, debuffer, etc.) that supports an Ada Capability on any embedded computers such as the MIL-STD- 1750A or MIL-STD- 1862. Until the AIE effort addresses this problem the AIE will never be an important Ada tool for real-time embedded computer applications.

	1	B

Where are the contractual statement of work? Where is the users manual? Without these documents, it is difficult for the reviewer to know whether the type A specification responds to the customer's needs.

	0	A

I feel that the overall requirements of the AIE are well thought out except I believe the space permitted the compiler in the minimum configuration is excessive and will eliminate several otherwise suitable hosts. The Intermetrics proposed design is equally as thorough and ignoring performance considerations, should provide a comprehensive, modern system. There are insufficient details at this time to determine whether the system will be efficient of resources and unfortunately this will have as much to do with system acceptance as reliability and comprehensiveness.

# 2.1	3	B

The reference to the manual for Ada should be changed to reflect the January 1983 date for ANSI Ada.

# 3.1.1	5-7	A

The reasons for communications between different KAPSEs, each of which can serve a number of users, should be given.

# 3.1.1.1	5	B

The 7 major subsystems in this subsection appear to be significantly smaller in scope than those requested by the original phase I statement of work for the AIE.

# 3.1.1.3	6	B

Is the recompilation that is referenced in this section done automatically or does the user have any control of it.

# 3.1.1.5	6	B

This paragraph should state whether or not there is a full screen option for the editor.

# 3.1.1.6	7	B

Simulation/emulation implies that the AIE will support target computers other than the hosts. Would it be appropriate to list potential targets in the System Specification?

# 3.1.1.7	7	B

The subsystem in this section is not clearly explained. I don't know why it is needed or what it supports.

# 3.1.5.1	8	

Is it a wise design decision to give each program its own run- time system? Won't this have a deleterious effect on overhead? Is there code that could be shared?

	12	

The paragraph states that the "KAPSE itself is an Ada program, using a specialized version of the Ada run time system ..." Other than the brief note about this specialized version that appears here in the System Specification, there does not seem to be any explanation of the functions performed by the KAPSE's RTS. In the KAPSE B-5, the RTS that is specified seems to be the one used by all other Ada programs.

	8	B

This paragraph states that all embedded machine code will be contained in the KAPSE. Will the facilities for generating machine code be made available to MAPSE users for future applications requiring machine code generation?

# 3.1.5.1.1 12 B

The second paragraph is confusing. It is not clear whether the users interface is to the KAPSE or to the vm/sp.

# 3.1.5.1.2 13

This seems to be a better architecture than the 4341. There is only one KAPSE.

# 3.1.5.2 13 B

A reference to Figure 6 would be most helpful.

# 3.1.5.6.1 12

The architecture chosen may caused response time problems. Giving each user a separate KAPSE on a separate virtual machine appears to be two levels of operating system overhead.

# 3.1.6 14 B

Is this list of GFE to be construed as a minimal set of hardware for running the AIE If not, what is the minimum Also, what is the minimum Perkin-Elmer configuration?

14 B

Why is there no description for the Perkins Elmer 8/32 configuration?

# 3.2.1 (2) 15

Embedded computers are alluded to but not specified. This is really part of a general comment - it is understood that the AIE will, someday, be used to develop operational Ada Software for embedded computers. However, other than indirect references such as this there is very little in the way of concrete support for targets other than the host.

15

Is debugging strictly limited to Ada statements and symbols? Will there be means to look at objects in internal (hex or binary) format?

14 B

To someone who is not familiar with the IBM systems the change in terms that are used to refer to the IBM host configuration is very confusing. This section contains an example of this problem. The term CP-370/VM is unknown to the author. How is this related to previous references to VM/SP?

3.2.1.1 16

The speed criterion is based on one compile and three users doing interactive editing or debugging. This is a light load and does not show when the system becomes more heavily loaded. There should be a set of time specification for 2..MAX_USERS doing simultaneous compilations.

16 A

It is not clear which of the size characteristics are inclusive and which are additive. It appears that summing sizes 1,2,4,5, and 6 indicates that to perform a compilation may actually require 848KB. I sure hope that I am wrong.

# 3.2.1.3 (8) 19

Reliability is specified in general terms only. Is it possible to be more specific?

19

The goal "Maximum use of reusable portions" is not fulfilled when there is a separate KAPSE for each user.

19 B

The second paragraph on applicability states that the AIE should be applicable to the development of embedded computer software for DoD projects. I see nothing in this type "A" specification that addresses this point!

In Paragraph Five the word "lightly" should be replaced with "loosely".

# 3.2.9 21 B

Does the third sentence of this paragraph imply that the AIE will never be transported to anything but 32-bit architecture main frames?

# 3.3 22 B

Will this document (CPDP), or some reasonable subset of it be made available to the AIE reviewers From the context in which it is referenced, and other contexts, it may have technical information of interest to the reviewers.

3.3.5.9 22 B

I thought the interface to the back end was the intermediate language BILL.

# 3.7.1 26 B

The first paragraph seems to summarize the function of the KAPSE. It only talks about the functions regarding the database. Why aren't other functions summarized in this paragraph? The nomenclature KAPSE/database is used quite often throughout this specification. Why? The term KAPSE implies that it has a database. The term appears to be redundant.

# 3.7.3 (5) 31

What specifically is a "near-executable memory image"? Is it non-executable only because it has not been loaded into virtual memory or is there further processing required, beyond that of loading it?

28

Is a stub just a return or is it a message plus a return This should be specified. A message plus a return is, of course, more meaningful and useful.

31

Is the PIF also the back end of the compiler? It also accepts Diana and produces executable code!

28 B

Is the recompilation that is referenced in the paragraph marked 1 done automatically or is it user initiated?

# 3.7.4 (2) 31

It should read "dividing the total number of statements by CPU time"

31 B

In the paragraph marked 3 it refers to the DIANA intermediate language. Instead of DIANA a better choice would be the term IDL. DIANA is defined using IDL.

# 3.7.5 33

Couldn't there be both a line and a screen mode? IBM does offer a screen editor with the 4341 (IBM XEDIT).

33 B

Will the output data from the tools invoked within the editor be displayed or will they be appended and inserted into the files that are being edited?

I think you really want to refer to this tool as an editor tool and not as the edited tool.

# 3.7.6 33

There should be a facility to display internal format (for example, a dump option).

# 3.7.7 35

A number would give a lot more meaning (and testability) than general statements like "far in excess".

# 3.7.8 35

Are the "secondary MAPSE tools" part of the deliverable AIE?

35

What specifically is an "object module converter"?

B

Why isn't the bootstrap compiler a deliverable item? Such a tool would be useful in the interim before the full Ada compiler would be delivered. Are the other tools that are referred to in the Paragraph two deliverable?

# 3.8 35 B

Is this section responding to a contractual requirement? Without a statement of work it is hard for this reviewer to determine just how well this section responds to customer's needs.

# 4.1.1.2 39 B

Does this section imply that only 100% compliance to the ACVC tests is acceptable to the customer? Is less than 100% compliance acceptable to the customer?

# FIGURE 1 9 B

The figure doesn't seem to correspond with some of the interfaces that were described for the KAPSE overview. It would appear that there should be a bi- directional line going from the box marked "Users and Peripherals" to the box marked "Host Machine Plus Necessary Software".

There also appears to be a bi- directional line that needs to go from the box "terminal IO" to the box marked "host IO". Furthermore there are some minor errors or corrections that should be made. I/F in the Tools" Communications" box needs to be defined. Also the box marked "Database" should be defined as the KAPSE.

# FIGURE 2 10 B

The box marked "Prog Int Fac" should read "Program Initiate Facility".

# FIGURE 3 11 B

The figure is not clear. It has no explanations of the symbols. It takes careful examination to determine that an oval is a data base object and a rectangle is a tool. What does a dotted line mean? Some dotted lines go off into space.

11 B

It is not clear how the boxes marked "Compiler","Linker", and "Program Library Manager/VMM" fit into the context of the diagram. It is suggested that somehow these boxes be marked as running Ada programs.

# FIGURE 9 32

There should be separate back ends for producing emulated target code and actual target code.

!SPECIFICATION TITLE: ADA COMPILER PHASES
!SPECIFICATION DATE: 82 NOV 5
!SPECIFICATION NUMBER: IR-277-2
!REVIEW DATE: 83 MAY 1

Paragraph	Page	Competence Level
# 0.0	1	A

The specification introduces many new terms unfamiliar to the reviewer. These include terms such as "library modes" in section 3.2.4.2 and "domain" and "sub-domain" in sections 3.2.4.4.2. It would be most useful if this specification had a glossary that defined all new terms introduced in this specification.

	1	A

Without a statement of work to reference it is impossible to offer any comments on how compliant the specification is to the statement of work.

# 2.1	5	A

Change the reference for the reference manual for Ada from July 1982 to January 1983 to reflect the new ANSI Standard.

# 3.2.4.4	10	A

It is unclear to this reviewer what purpose the virtual memory methodology interface serves. There is just enough detail in this section describing the virtual memory methodology to confuse the reviewer.

# 3.2.4.6	12	A

The default value for NOCODE should be zero. Fifty is a reasonable value for NOSEM, but code should never be produced for invalid programs unless specifically requested. Also the value n should apply to the sum of syntactic and semantic errors.

Incidentally, I assume that the compiler will never insert code into the program library after a FATAL or INTERNAL error, but will always generate code no matter how many WARNINGs or NOTEs were generated.

	12	A

The LIST default should be ON. There are too many UNIX alumni on this project. Seriously, novice users may never discover why their programs fail so badly that they never even get a listing, but expert users will normally use scripts or abbreviations.

11 A

In paragraph 4 of this section it states that if the library parameter is omitted the compile request is interpreted as a request for a syntax check with no semantic processing. This seems like a rather awkward way of specifying this request. A more straightforward way would be to have an option that states specifically that the user wants a syntax check only. Furthermore as this paragraph is currently stated it prevents the definition of a default program library which would be very useful to most users of the AIE.

The discussion in this section regarding the LIST options is confusing. In particular the interaction among the options ON/OFF SOURCE/NO SOURCE. What happens when the user specifies that listing is to be ON and there is NOSOURCE?

In the paragraph that describes the DEBUG option will the ALTER and BREAK directives prevent the compiler from performing code movement and dead elimination?

In the paragraph describing the REORDER option it is not clear why the compiler needs to reorder the compilation of units?

11 A

The effect of compiling a program with no library specified should be to perform syntactic and semantic checking using only those library units which are specified in the LRM. Since a compilation can contain many compilation units this type of checking can be used for final check-out of systems before shipment to insure that all necessary source is included. (And in the proper order.)

# 3.2.4.9 13 A

It is not clear why the compiler would be invoked by the program library interface to assist in updating the library.

# 3.2.5.3 20 A

This section states that several of the back end phases perform machine independent optimizations based on machine independent cost criteria or cost functions. Where are these functions defined? What factors are used in calculating these cost figures?

# 3.3.1.2.2 24 A

Subsection A describes new attributes that are added to the abstract syntax tree. Why aren't the attributes for the partial symbol table and the name table described in this subsection? Both of these structures are produced by the first phase of the front end.

Subsection D of this section describes a list of pre-semantic checks that are performed. It is stated that none of these checks require access to symbol information, i.e. the symbol table. How then can semantic Check 3 for analyzing private or incomplete typed declarations be performed without access to a symbol table?

# 3.3.1.2.4	29	A

This section describes several alternatives to the problem of creating two different trees in the front end. One tree is the Abstract Syntax Tree; the other is the DIANA tree. Which of these alternatives has been chosen? How do each one of these alternatives affect the rehosting of the front end?

# 3.3.1.3.2	42	A

Where is Subsection D?

	37	A

How is the symbol table related to the list of IDL ITEMS in the DIANA tree? Is there one entry in the symbol table for each ITEM or ID defined (in the items of the DIANA tree)? Is the symbol table a completely unique and separate DIANA data structure or is it dispersed and distributed among the ITEMS nodes that are found in the DIANA tree?

# 3.3.1.3.2.1.2.1	49	A

In the second paragraph, third sentence, scratch the phrase "as these phases determine the feasibility of code sharing". It is repeated twice.

# 3.3.2.1	49	A

This section states," that the phase GENINST determines if instantiations can share generic implementations that have been generated for a previous instantiation of a given generic". The specification states that in general the previous phase SEN is unable to determine that multiple instantiations can be shared, because it's unaware of the run time representation of the data types. It is not clear to the reviewer what knowledge has been obtained between the phases SEN and GENINST that enables the phase GENINST to work?

# 3.3.2.1.1	50	A

How can the phase GENINST add information to the DIANA tree about instantiations yet preserve the original DIANA tree?

# 3.3.2.2.2.6	53	A

The phase STATINFO produces some very useful attributes only when the option LIST XREF is given. These attributes that information, such as, all references to defined ids, a list of calls made within a procedure and a list of all external references. Why wouldn't this phase produce these attributes anyway since it appears that they are most useful for later phases of the middle and back end of the compiler?

# 3.3.2.3.2.1 57 A

Are the descriptors described in this section a compile time or run time data structure?

# 3.3.2.3.2.1.1.2 59 A

What is a "ground type" that is referenced in the last sentence of this section?

The second paragraph states that run time data structures that compute the attributes IMAGE, VALUE VAL POS, SUCC and PRED for enumeration types when those types are given in a representation clause. They are also needed when the type is used in IO or when objects of those types are used in connection with the image or value attribute.

# 3.3.2.3.2.2 62 A

Why are the data objects of packages, subprograms, etc. determined so early in the compilation process? Why isn't this layout done later after live/dead object lifetime analysis is performed? Laying out the objects for subprograms and packages before the live dead object lifetime analysis deprives the compiler of the opportunity to overlay objects with disjoint lifetimes. The live dead object lifetime analysis that is done later in the back end by the phase TNBIND will only optimize the temporary locations not the other objects within the sub- program or packages.

# 3.3.2.3.2.2.2.2 64&65 A

This section states that the storage associated with sub program frame is divided between the primary stack to hold static size data and secondary stack to hold dynamically sized data. How well will this design feature transfer to other target architectures besides the IBM 360/370?

# 3.3.2.3.2.2.3 65 A

What is a "region of the array" that's referenced in the last sentence of this section?

# 3.3.2.3.2.2.5 66 A

The last sentence of this section is confusing. It should read "in processing a generic instantiation, STORAGE need only instantiate a copy of the prototype from the table that parameterizes any shared code body used by that instance." What is this table that is referenced in this sentence? Is it the list which the phase GENINST generates for each generic template (see section 3.3.2.1.2)?

3.3.2.3.4 66 A

How was the 15% figure referenced in the last sentence determined?

# 3.3.2.4.2.6 72 A

In this section it states that storage for resources declared in a package are found in the frame for the enclosing unit. If the package is a library unit, what then is the enclosing unit for those library packages?

# 3.3.2.4.2.7 72 A

The last sentence of this section implies that whenever an object is renamed the address for that renamed entity is always stored in a separate address location. Why is that necessary if the address of the object is always static? Why not replace the name of the renamed entity wherever the new name appears in the code.

# 3.3.3.3.2 85 A

The subsection A on live dead analysis does not include any discussion of exception handlers. Due to the spurious nature of exceptions any object that is referenced within an exception handler should never be put into a temporary such as a register or spill area.

# APPENDIX A.3 104 A

Replace all uses of the word BAD with INVALID. The use of words such as BAD, ABORTED, and FATAL should be avoided in error messages if you intend to interface with people. The use of BAD is the only such mistake here. (Assuming that only the developers ever see FATAL errors. The compiler should not be released if it still contains such errors reachable under normal circumstances. Of course, if a disk or memory board fails, some swearing by the compiler can be tolerated.)

!SPECIFICATION TITLE: AIE: COMPILER PHASES B5-AIE(1)-COMP(1)
!SPECIFICATION DATE: 5 NOVEMBER 1982
!SPECIFICATION NUMBER: IR-677-2
!REVIEW DATE: MARCH 1983

Paragraph	Page	Competence Level
# 0.0	0	A

More information is required regarding the built-in maintenance facilities of the compiler to determine both its maintainability and its adaptability to new host and/or targets.

# 3.2.4.6	11	

A general concern, which does not appear to be addressed in the compiler B-5 specification (but which may be answered when the PIF B-5 becomes available) should be raised now.

The issue relates to the inclusion of actual instantiated code being included in the source listing from a compile. In other words, when a generic unit is instantiated in a MAPSE users program, will he see the actual code he created in his listing, at the point at which he caused the instantiation to happen. The answer does not seem to be in this document.

For the sake of both testing and maintaining a program, it is most desirable to have this capability. It appears that something like this may be recreated, after the fact, by a tool called the source code reconstructor, also contained in the PIF. However, this may, or may not, be adequate from a maintainers viewpoint.

	11	A

Permitting the source to come from the standard input file supports a mode of compiler execution wherein a large program, i.e., the Ada compiler will be paged in and out at user input speed. This is an inefficient use of computer resources.

	12	A

The most difficult bugs occur ~~not~~ during program check out but during production use when a large volume of data is available to create the combinations and circumstances that expose boundary conditions and interrelationships that all too often do not exist in test cases and validation suites. To require hooks and suppressed optimization to permit effective debugging indicates an ignorance of the requirements of the embedded computer application area.

	13	A

The rationale for source reconstruction from DIANA escapes me. What's wrong with the original source. Descriptors which correlate the DIANA with the contributing source files seem sufficient. To include source into this already space-consuming data base aggravates a problem that won't be fully appreciated until the first large system uses this compiler.

13 A

Finer control of optimization than simply to favor time, favor space, or none on a program wide basis is desirable. Certain types of optimizations, particularly those related to loops, can have very different effects depending upon loop frequency and content. Regional control and pragmas which provide additional program information such as loop frequency or the expected ratios for if and case statement paths can provide significant program improvements with little impact upon compiler implementation costs or compilation performance.

14 A

Cost-effective compiler check-out and maintenance requires more control of the compiler's internal maintenance facilities (which I hope is more than just TRACE) than on and off.

# 3.3.1.2.4 29 A

This requirement for compiler-produced external AST and DIANA files is controlling compiler design and performance. It would seem that if the compiler developers need a more efficient representation than tool programs could use one also.

# 3.3.1.3.4 49

The paragraph states that it may be necessary to place some limit on the size of a separate compilation unit in order to attain a compiler speed objective. This is acceptable, if this (unknown) limit is used only as a number in the acceptance testing. It will NOT be acceptable if some arbitrary unit size is wired into the compiler, and serves as a constraint on future applications developers.

# 3.3.2.3 55

The entire section on STORAGE does not address the handling of representation specifications, which should be a major special requirement. The fact that this topic is omitted may be confirmed by referencing Appendix A, p1Ø3. This appendix lists the compiler error messages. It will be noted that the appendix states that STORAGE will generate an error message for bad representation specifications. Since the specification on STORAGE does not appear to mention representation specifications, it seems as if there is an inconsistency in this B-5, and the document is incomplete in this area.

# 3.3.2.3.1.1.4 59

The specification states "all fixed point types are a single word long". This is not a good ground rule, if one considers transportability, especially from the viewpoint of real targets. As long as this compiler generates code for the IBM and P-E targets, with their 32 bit words, this is likely to be a safe implementation. However, in the embedded world, there are targets with 16 bit words, and there is very definitely a requirement for extended precision fixed point arithmetic. This restriction is not one that could be lived with, when military targets are considered.

# 3.3.2.4.4 72

The timing requirement stated in this paragraph is hedged with the statement about the tree part of DIANA not being paged. This restriction may make the whole speed issue somewhat less meaningful. It also gives rise to another issue - the paging facilities of VMM. In section 3.3.1.1.2, VMM is presented as a useful device that isolates the detailed bookkeeping of paging from the actual MAPSE tool. However, in many of the special requirements section, such as 3.3.2.2.4 (p 55), the speed requirement is hedged with a statement that paging, if it occurs, will cause problems in terms of meeting the speed objective. It appears the VMM system has to be overridden in certain cases, or workarounds established. These tend to cast some credibility on the overall usefulness of the VMM.

# 3.3.3.1.2 (D) 77

It would not be proper to replace an expression A*Ø by 1.

77

Applying strength reduction to floating variables is not generally safe, as roundoff errors in a series of additions tend to accumulate. As a drastic example, consider:

```
      .
      .
      .
      .
      .
          lambda:=1.Ø;
          while lambda +1.Ø/=1.Ø
          loop
               lambda:=lambda/2.Ø;
          end loop;
               -- make lambda smallest negative power
               -- of two such that 1.Ø+lambda=1.Ø
          x:=Ø.Ø;
          count:=Ø;
          Z:=1.Ø+4.Ø*lambda;
          loop
               count:=count+1;
               x=x+1.Ø;
               y:=1.Ø+x*lambda;
               exit when y>z;
          end loop;
          put (count);
```

```
count:=0;
x_prime:=1.0;
        --remove loop invariant 1.0 from y,
        --and add initial value 0.0*lambda
loop
    count:=count+1;
    x_prime:=x_prime+lambda;
    exit when x_prime>z;
end loop;
put (count);
    .
    .
    .
```

The second loop involving count is derived form the first by strength reduction,constant folding, code motion and dead variable elimination in a straight-forward way. However, it will never terminate (unlike the first) since the statement "x_prime:=x_prime+lambda" will actually leave x_prime unchanged from its initial value of 1.0. It is in no reasonable manner equivalent to the first loop. Strength reduction on floating point variables can sometimes be numerically justified, but we should rely on programmers to write the code that way if desired, since the automatic transformation is not safe. It is, however, quite justified over integer variables, and is quite effective in simplifying subscription expressions.

77 A

I assume that the algebraic simplification is either A**0 or the replacement is 0 rather than 1.

# 3.3.3.5.3 94 A

What is the hook table? Neither the compiler specification nor the debugger specification clearly describes what it is or how it differs from the statement table.

# 3.5 96 A

It is not clear whether the KAPSE size is included in the 300KB. If not this minimum size is most certainly too large. It is undesirably too large in any case to maximize the number of potential hosts.

98 A

Although 2000 lines should be ample for most programs giving current programming conventions, this is too small as an absolute limit for a compilation unit particularly since this includes packages.

97 A

Given the intended application environment of Ada, it would seem that code efficiency should precede retargetability/rehostablility as a priority. This is particularly true with the current philosophy of utilizing standard instruction set architectures.

!SPECIFICATION TITLE: KAPSE/DATABASE
!SPECIFICATION DATE: 82 NOV 12
!SPECIFICATION NUMBER: IR-278-2
!REVIEW DATE: 83 MAY 1

Paragraph	Page	Competence Level
# 0.0	GENERAL	B

It should be noted that I did not undertake a detailed analysis of the security properties of the database system described in this specification. My gut feel is that there are no obvious "holes". I would hope a more detailed analysis of the security properties of the design (given more extensive design information) would be conducted.

# 3.2.2	10	B

The design goal to provide support for thirty 300 baud input streams is too low if the corresponding number of 1200 baud streams is 7.5 or 9600 baud streams is one. User access to the AIE is more likely to be at a rate greater than 300 baud than not.

# 3.2.4.1	11	B

The organization of the interfaces here is in some ways questionable. In particular, it would seem ill advised to include the mail_system with the program_invocation facilities; obviously the later is mandatory and fundamental to the entire APSE, the former is not.

# 3.2.4.2.1	12	B

It is stated that each component of the tools composite object is an executable context. Where do the source, documentation, etc. for these executable contexts reside?

Does this section prohibit an AIE installation from having more than four components in its root object?

	12	B

It is not clear from the description given in this section whether the conve..tions indicated are recommended or mandatory. The very fact that some conventions are mentioned opens the possibility that some components of the KAPSE may come (either now or in the future) to depend on these conventions.

The conventions stated are not specified in sufficient detail to form a complete set of practices for structuring the database. A more elaborated set, for example those associated with the Multics system, should be presented if any conventions are presented at all.

	12	B

The notation used in the last paragraph on the page is not previously explained; a forward pointer is needed.

# 3.2.4.3.1 14 B

The access_method is implied to be permanently associated with an object. Is it in fact true that a single object may, over it's lifetime, be accessed by only one access method? If so, this may be overly restrictive; certainly "escape" into some access method that is simply a byte stream may be necessary. If not, then restrictions on using different access methods at different points in time is not explained.

14-15 B

As written, this section is more complex than necessary. A top down description that starts with a discussion of extended objects might be an improvement.

This concept of "extended window object" is discussed without describing a window object which is not extended.

# 3.2.4.3.2 16 B

The possibility of attribute values being files or windows in not explained. I cannot find a single explanation of this option in the remainder of the document. It is an interesting, and potentially useful, feature - it should be treated completely.

# 3.2.4.3.3.1 20 B

The notation "'PRIORITY" is used before it is explained; a forward pointer is needed.

# 3.2.4.3.3.2 24 B

How does the syntax for a pathname which is expressed relative to the root differ from the syntax of a pathname which is expressed relative to a local window?

How does the pathname SYSTEM.PRINT'QUEUE'FIRST.BODY differ from SYSTEM.PRINT.QUEUE.FIRST.BODY

# 3.2.4.3.4.2 22 B

The explanation of CATEGORY and CATEGORY_DESCRIPTOR is particularly unclear. It is not evident why both are needed. Apparently the CATEGORY takes on values that are ASCII strings, and hence it is not a descriptor of the actual contents - which is the purpose of the CATEGORY_DESCRIPTOR.

I attempted to examine the entire specification quite throughly to resolve this matter. Unfortunately, without a comprehensive index, the attempt was frustrated.

# 3.2.4.3.4.4.1	24	B

The note on the bottom of the page needs elaboration. I suspect it conveys important information, which I could not fully understand. Examinations of the remainder of the specification did serve to clear up this matter.

# 3.2.4.3.5.3	26	B

The facilities described in this paragraph are particularly valuable; their inclusion is to be commended.

# 3.2.4.3.6.5	31	B

The set of role modifiers specified in this paragraph appears to be sufficient, and complete from a practical viewpoint. I question the validity of both of these statements in as much as I did not undertake a detailed analysis of the role facilities of the KAPSE.

# 3.2.4.3.8.2	33	B

The periodic "garbage collection" of window keys will have some performance impact on the KAPSE as a whole. The extent of this performance impact is not apparent, although it may be significant.

# 3.2.4.3.9	36	B

It is not clear whether proper consideration has been given to recreating an object from the "script". In particular, if the program that effected the state transformation in fact invoked other programs, then the closure of all input to all programs so invoked must be recorded. It is not clear whether the KAPSE will have provisions for detecting such occurrences, and recording appropriate information in those cases. In theory, it would be trivially possible to write programs whose operations could not be repeated - for example a program whose function depends upon the date. In practice such programs may not be encountered, but there are a wealth of difficulties that may be.

# 3.2.4.4	37	B

I most strongly object to the organization of the KAPSE that results in host dependent details being "hidden" within the SIMPCOMP component. A separate component should be defined that contains all host dependencies that are associated with both terminal and mass storage i/o devices.

# 3.2.4.4.1.1	39	B

The inter machine (i.e. KAPSE) communications seems to be largely TBD throughout. This is an issue which should not be treated lightly. It may turn out that it is desirable, for a number of reasons, to run multiple AIEs on the same physical machine. At present, it would seem debatable whether adequate facilities will be provided in the AIE to allow this.

# 3.2.5 41 B

During the first reading I was absolutely shocked when I began this section. In section 3.2.4.3 I had discovered a great wealth of information about the structure of the database; with the overwhelming flavor being high level facilities to be provided. In the beginning of this section the mention of block i/o routines took me quite by surprise.

The figure 3-8 (pg 42), for example, does not indicate what other components of the KAPSE the SIMPCOMP CPC interfaces with. Comparing this with figure 3-1 does not resolve the difficulty. In fact, it would lead one to believe that SIMPCOMP is partially in the database portion of figure 3-1, and partially in the KAPSE/Host portion of figure 3-1. Such a conclusion immediately leads one to question the portability of the entire AIE (and the KAPSE in particular) in as much as host dependencies may be spread throughout the KAPSE.

# 3.2.5.1 43 B

I would strongly recommend that the device io and block io components of the SIMPCOMP CPC be removed and placed in a separate CPC that is identified as host dependent. The remainder of the SIMPCOMP components should then be host independent.

# 3.2.5.3 46 B

The same blurring of levels of abstraction seems to occur in the architecture of the MULTPROG CPCI. Clearly, mail terminal screen management, and program loading are very different kinds of facilities. I very much dislike a packaging or design that mixes applies, oranges, and elephants. This is what has happened here; at least to some extent. The mail and terminal management should be in a separate CPCI.

# 3.2.5.3.4 47 B

Note that KAPSE-KAPSE communications is here completely TBD! That is clearly unacceptable.

# 3.3.1.1.2.3 52 B

The parameters of the buffer management algorithm(s) that may be tuned should be specified.

# 3.3.1.1.4 53 B

The "virtual copy" facility seems both well justified and well thought out; it may prove essential to obtaining acceptable performance in the

AIE.

# 3.3.1.3.1 61 B

The provision for data clumps would likewise seem both necessary and well conceived.

# 3.3.1.3.2 63 B

More details regarding the various access methods is necessary in order to fully evaluate the functionality to be present in the AIE. In general, the access methods provided appear quite reasonable. I question in particular, the results (or possibility) of applying various access methods to the same object; the information provided leaves that an open question.

# 3.3.1.4 67 B

The specification of the operations in the package DIRECT_ACCESS need to be elaborated. The information provided seems to not tie together all the concepts, etc. introduced in 3.2 to the degree I would prefer; much of the "tying together" is left to the reader.

# 3.3.1.5 76 B

Same remark as on section 3.3.1.4. Here the situation is even more extreme.

# 3.3.2 79 B

This section is much more complete than sections 3.3.1.4 or 3.3.1.5, although still not "perfect".

# 3.3.2.1.4 85 B

The window traversal "cache" would seem necessary; inclusion is certainly well advised until more performance analysis information is available.

# 3.3.2.3.2 93 B

Is this the first mention that access control facilities are used to accomplish synchronization? I believe it is. At any rate, a more prominent discussion of this matter should appear in section 3.2.

3.3.3.7 119-121 B

The specified mail system is inadequate for general use in at least the following ways:

o it lacks necessary operations such as delete, reply, forward,...

o there should not be a me-to-me correlation between users and mailboxes

o it is not clear that the facility allows users to share mailboxes

o the creation of a mailbox should not require intervention by the system manager

However, the high level tool for manipulation mail belongs in the MAPSE rather than the KAPSE. The KAPSE need only provide the facilities needed by the mail system, and it is likely that the database already does. If this is the case, then this section should be deleted.

# 3.3.5	132	A

There is no global view of the run time system described in the specification; that is there is no relationship described between a main program, its library units, and its subunits. This would be useful in giving the reader some context to understand where different portions of the run- time system such as the unit execution support or the storage management fit.

# 3.3.5.1.1.2	133	A

This section fails to describe the generic subprogram parameter descriptors.

# 3.3.5.1.1.3	134	A

What is the unit type that is described in the unit data area?

# 3.3.5.1.1.4	134	A

How are the registers described in this section mapped onto the real register set for the IBM 370? Are all these registers required at once or is some subset of them required at all times?

# 3.3.5.2	132-138	B

The likely effect of the heap protocols is to cause programmers to use representation specifications to specify memory pools for access types for more often than intended by the Ada language designers. It should be possible to implicitly cause the deallocation of memory associated with an access type on exit of the scope without specifying either a representation specification or a pragma.

# 3.3.5.2.2.2	140	A

Forcing data objects to reside completely within a single storage segment of 4,096 bytes is unjustifiably restrictive. Clearly there must be some way to allocate very large data objects on a secondary stack. Languages like PL1 and Fortran for the IBM360- 370's don't have these unnecessary restrictions then why should Ada.

# 3.3.5.2.2.4	143	A

How does storage management keep segments reserved for heaps storage separate from segments reserved for dynamic storage for subprogram frames?

# 3.3.5.3.1.5	146	A

Where do the headers for the dependency list described in this section reside?

# 3.3.5.3.2.3	158	A

Tasks are not added to the appropriate activation list during task activation as described in this section. They are added to the appropriate dependency lists when the task is created.

# 3.3.5.3.2.5	159	A

The first reference to the term "acceptor" in the third sentence of the third paragraph of this section should be changed to "caller."

# 3.3.5.4.1.1	163	A

How do you insure that each 32-bit exception identifier is unique across compilation units?

# 3.3.5.4.1.3	163	A

The parameter RET is missing from the list of registers to the operation PRCALLER.

# 3.3.5.4.2.5	166	A

How do you know when an exception is propagated out of the main program?

# 3.3.5.6.2	170	A

What is the criteria that determines when it is appropriate to insert inline the code for the type support routines?

!SPECIFICATION TITLE: KAPSE/DATA BASE B5
!SPECIFICATION DATE: 12NOVEMBER 1982
!SPECIFICATION NUMBER: IR-678-2
!REVIEW DATE: 15 APRIL 1983

Paragraph	Page	Competence Level
# 0.0	0	A

The KAPSE and database services appear to be complete. The proof of the viability of the system lies in its performance characteristics which can not be evaluated at the B5 specification level.

3.0 5

This document that not appear to contain any timing or sizing data. Are there budgets established that the KAPSE will be measured against.

# 3.1.1 5

"In so far as possible" is vague wording. It is a design goal and not a requirement.

# 3.1.1.4.1 69

Reading a record takes two operations: SET_OFFSET and READ. It would be more convenient to have a single READ procedure with the record key as a parameter.

# 3.2.1 8

This section says that the peripheral suite of the IBM 4341 must be supported in a host independent manner. This is an extension of the meaning of "host independent". Must any CPU support 4341 peripherals?

# 3.2.4.1 11

Every call to the KAPSE is a message to an agent task. This may degrade response time.

11

The paragraph states that services are provided to an Ada program via packages linked into the running program.

In what manner will this be accomplished. Will a user program just receive packages corresponding to those functions his program either directly or indirectly calls. Or will the user be burdened with the overhead of an entire KAPSE for every running program Or will the answer lie somewhere in the middle - will a user program have a whole CPC linked into it if he uses one of its services.

11

Is there supposed to be a 1-1 correlation between the packages delineated in this paragraph and the list of CPCs contained in the Systems Specification, page 49. Which list is correct

# 3.2.4.3 38

The 24 pages describing the Database-Tool Interface describe an very complicated system. Other operating systems (IBM OS, CMS, TSO) manage data in a much simpler and easy-to-understand way. The interface with Secondary Windows, Role Translations, and Transitory Windows could require as much training as the Ada language itself.

# 3.2.4.4.1.1 38

The architecture of the IBM 4341 version gives each user a KAPSE running in a virtual machine. Since the KAPSE run time system is time sharing (processes and tasks) on top of the VM/SP time sharing of virtual machines, the response time will be degraded.

# 3.2.5.3.2 46

Communication from program to KAPSE is by message. It might be better if it was by calls to system subroutines.

# 3.2.5.3.5 47

The list of control characters is incomplete and does not belong here. It should be part of the command language.

# 3.2.5.3.6 47

There should be a way to change a users privilege level without logout and re- login.

3.3.1 5Ø

The Section on Data Management would be appropriate if it was implemented on a bare machine. It has all the primitive functions of a file management system. It appears to be implemented on top of the VM/SP operating system, which may lead to excessive overhead.

3.3.1.1.1.2 5Ø

"The logically sequential blocks are allocated non-sequentially" — Does this mean that the blocks are allocated so as to purposely slow down the system? That is not efficient.

# 3.3.1.2.1.2 58

This section should be filled in.

# 3.3.1.4.4.1 72

Procedure NO_ECHO has a syntax error. The word "procedure" is repeated. The system would be more user-friendly if SET_ECHO and NO_ECHO were one procedure with a Boolean ON-OFF as a parameter.

# 3.3.1.4.5 75

This package gives Ada a somewhat awkward FORTRAN style I-O capability. There should be a better way to specify formatted I- O.

# 3.3.2.3.2 94

The package ACCESS_SYNCHRONIZATION provides complex access control but is not clear on whether it provides file level locking or record level locking. This should be clarified.

# 3.3.3.1 97

Comment line 11 of the procedure specification of LOAD_PROGRAM contains the statement "extra effort will be made to share code..." Is this to be considered as a requirement

In the same general area, p98, paragraphs 3.3.3.1.3, and 3.3.3.1.4 both contain the statement "when sharing is warranted...", which leads one to believe that some tool exists (or will exist to track frequency of use of KAPSE programs. Is that a reasonable assumption Where will that tool reside Is it to be contained in the KAPSE or provided by some other CPCI.

# 3.3.3.3.1.1 105

With regard to package PROGRAM_INVOCATION, function CALL_PROGRAM, is the parameter PROGRAM_PATH the same thing as the parameter LOAD_MODULE_NAME that was seen on page 97 If so, could one of them be changed to agree with the other

# 3.3.3.3.2.1 108

In function PICK_PARAM, the formal parameter PARAM_NAME should be specified with a default.

# 3.3.3.6.1.2 116

The login process should permit the execution of an optionally suppied, user generated script file. This seems to be implied rather than actually stated as a requirement.

3.3.5.1.1.4 134 A

The number of registers used for a call seems excessive. Are all of these dedicated throughout the subprogram body? It would seem that some of these could be eliminated by having indirect pointers from canonical locations in the caller's local frame.

# 3.3.5.1.2.1 135 A

It is often better to perform register saves within the callee in one place for those registers used, than to perform saves at the point of each call which is being done for the floating registers.

3.5.Ø 132 A

The run-time conventions appear to be based upon the 4341 characteristics. Potential target computers may not conform nor have the register resources afforded by this host. I trust that the design does not preclude adaptation to a wide spectrum of machine characteristics.

# FIGURE 3-2 13

Figure 3-1 is titled on the bottom. Figure 3-2 is titled on the top. There should bi consistency.

!SPECIFICATION TITLE: AIE: MAPSE COMMAND PROCESSOR B5-AIE(1)-MCP(1)
!SPECIFICATION DATE: 1 DECEMBER 1982
!SPECIFICATION NUMBER: IR-679-1
!REVIEW DATE: MARCH 1983

Paragraph	Page	Competence Level
# 0.0		C

Access rights were not addressed. I assume the access privileges on the objects submitted to and created by the MCP will be sufficiently discussed in the data base manager.

C

We should take advantage of the structure of the MCL and provide a syntax oriented MCP (indenting lines to the proper block structuring level).

C

I'm not sure if this is the appropriate place for this (maybe within the terminal driver within the KAPSE). But a good command editing facility should be provided. e.g. redisplay (and edit) the last command line entered.

Ø A

The overall capability of the MCP looks very comprehensive. One desirable feature appears to be missing (but may in fact be achievable by what is supported); this feature would permit the creation of a log of input commands, resulting output, or both without redirection from the standard devices.

#3.Ø

This "Requirements" section reads much more like a definition of an implementation of the MCP rather than a requirements or functional specification. The Appendix A specification of the MCL proper and the "main" sections must be better integrated and the implementation issues motivated.

# 3.2.4.3	8	C

If no general help or no parameter help exists a message stating such should be output rather than having the parameter attribute being undefined.

# 3.3.Ø	15-	A

I didn't understand the necessity for multiple tasks to process commands; these would seem to result in a non-trivial overhead.

3.3.1 11 A

I can see several places where the implicit semicolon rules will cause trouble. These could be "solved" by MCL restrictions or a precise set of disambiguation rules; but I think that the best solution is to add an explicit continuation symbol. The first example on page 12 implies that a comment is a continuation if it is not preceded by a semicolon. This would be very confusing to users.

Another solution is to decide that commands in compound commands should always end in semicolons (best if editing and reentry of previous commands is permitted) or that semicolons are required if a command is not followed by a reserved word which always follows a command such as ABORT, BEGIN, CASE, ELSE, ELSIF, EXIT, FOR, IF, LOGOUT, LOOP, RETURN, USE, THEN, WHEN. (Actually some of these reserved words can follow other reserved words which never terminate a command, for example OR ELSE.)

12 C

Does not mention the backslash (\) character is a line continuation character.

11-12 A

The rules for complete vs. continuation command lines are unclear and sound like continuation commands have to be broken very carefully to insure that a partial command is treated as a complete command.

Some provision is needed, if it doesn't already exist, to prohibit user interruption of an MCP_STARTUP execution. This will allow users to insure that security programs written to validate sign-on can't be circumvented by a issuance of a control-c.

# APPENDIX A.2.2 30 C

A minor point - but I don't understand why no whitespace is allowed between function name and the left parenthesis.

# APPENDIX A.6 33 B

I don't like the requirement that background tasks must not be running when LOGOUT or SUSPEND is executed. It may be that a DETACH command is required to permit compilations to continue while the user leaves the AIE environment, either to use the underlying operating system to check his mail, or to disconnect completely to go to lunch. I have had to baby sit a terminal too often to see this design flaw blithely accepted.

# APPENDIX A.8 35

The system should be careful about when it notifies users that a task has been completed. A user may well be executing another program which is controlling the screen. A completion message could easily cause important information to be lost (either over written or scrolled off the display) or cause confusion if it doesn't return the cursor to the position where it was when the completion notice occurred. They should only appear associated with prompts for command input.

# APPENDIX A.12.3 43 A

It sounds like database objects (which program context objects are) disappear if the exit status is "OK". If so, how can STATUS work for successfully terminated programs. If not, how long does this program context object remain in the system; is it deleted only by positive action of a user?

# APPENDIX A.16 45 C

Some syntax/semantics checking of the script file would be desirable before execution of the script.

# APPENDIX A.16.1 47 A

There must be a typo; I don't see how or why %COMPILE_ERRORS is created.

# APPENDIX B.1 (C) 49 A

The backslash character is missing.

49 A

The description of the use of the backslash ' ' character seems to conflict with other sentences regarding lexical units in the same paragraph, the description of continuation lines on page 12, the description of parameter passing in A.2 on page 27, and the description of expressions on page 52.

# APPENDIX B.3.4 55 A

Explicit conversion of a type to itself should be allowed. A user writing a script may not know what type of argument will be supplied and want to convert for safety, or to make the script easier to understand, or to guarantee that the result of an operation is of the type he expects. For example: REAL((1.7 + 6) * 1Ø)

Also, I assume that blank lines were omitted before and after INTEGER to STRING conversion.

TABLE A.4 4Ø

(and Section #A.12.1,page 41) It is not made clear whether %STATUS.EXECUTION.TIME refers to elapsed time or task time. For systems which maintain charging information, it would be desire to include a component "COST" analogous to EXECUTION.TIME.

!SPECIFICATION TITLE: AIE: PROGRAM INTEGRATION FACILITIES
!SPECIFICATION DATE: 22 MARCH 1983
!SPECIFICATION NUMBER: IR-681-1
!REVIEW DATE: 19 APRIL 1983

Paragraph	Page	Competence Level
# 0.0	0	A

The facility for separate libraries of utility programs (herein referred to as resource catalogs) is a necessary requirement for practical program development. I was glad to see that the AIE interpretation of the Ada Program Library accommodates such a facility. However, linkers have been traditional resource users and the mechanisms described in this specification imply even more overhead than normal. I think linker performance objectives that are within the user pain threshold should be established.

# 3.2.4.5	25	A

The implication is that both the AST and the DIANA representation of a program are retained in the program library. If the DIANA form is a further attributed AST as implied by the compiler specification, and the AST is reconstructable from the DIANA form as indicated by the DIANA Reference Manual, than I don't understand the need for both to be retained.

# 3.2.4.7.4	29	A

The assembly listing for the Perkin-Elmer 8/32 should correspond to the Perkin- Elmer Assembler listing format rather than the 4341 assembler.

# 3.2.5	31	A

Rather than retain the AST and DIANA around and require a source reconstructor tool it would seem to make more sense to retain the source around and build an elaborate AST and DIANA reconstructor. A good name for such a tool might be an _ST/_IANA _ttribute (or Ada) compiler.

# 3.3.1.0	32	A

I found it difficult to distinguish some of the terms used in this section and to understand all of the concepts and relationships presented.

# 3.3.1.1.1	33	A

Successful compilation of a program has never been an indication of program viability. Replacement of the backup with the compilation object prior to testing seems unwise. It not only destroys the previous backup but creates a duplicate copy of an untested object. If anything, why not replace the backup with the previous up-to-date object.

# 3.3.2.4.2 54 A

I suggest that a generated body should raise a PROGRAM_ERROR exception if it is entered.

# 3.3.3.6.2 65 A

How is an assembly listing generated if the information is not retained (ref. 3.3.3.6)?

!SPECIFICATION TITLE:	AIE: MAPSE DEBUGGING FACILITIES
!SPECIFICATION DATE:	5 JANUARY 1983
!SPECIFICATION NUMBER:	IR-682-1
!REVIEW DATE:	19 APRIL 1983

Paragraph	Page	Competence Level
# Ø.Ø	Ø	A

This entire facility appears to be based upon the assumption that machine level debugging will no longer be necessary. It sure would be nice if that were true but there doesn't seem to be anything in either Ada or the MAPSE that is going to obviate this need. Certainly there will be compiler errors even after passing the most rigorous of ACVC. Let's support the application programmer's needs. There should be some mechanism for register and hardware status display and relocatable memory dumps; let's not give up the tools that have worked in the past until the new ones are proven.

# 3.2.4.2	7	A

It appears that in order to use the debugger at all that the DSR and possibly the UTILITY PROCEDURES must be linked with the user's program. That implies that you cannot debug a production program without relinking. Boo!! Relinking can cause address sensitive errors such as wild stores to disappear. It should not be necessary to include any additional information in the execution image in order to support debugging.

# 3.2.4.2.1	8	A

The absence of hooks should not prohibit single stepping. All the information necessary to perform such execution appears to be present.

# 3.3.1.Ø	13,...	A

The debugger command syntax is too verbose. There is not a real requirement for these commands to be highly readable; debugger scripts almost never have to be maintained. The value of a debugger is measured by how quickly it will permit a bug to be revealed, pinpointed, and corrected. The debugger, like text editors, are the most frequently used programs during program development and should be responsive to the interactive user. I don't think this command syntax is.

# 3.3.1.2	18	A

The IGNORE command should permit specification of other than the current breakpoint.

	14	A

It appears that you can breakpoint all procedure entries or none. A list of procedures should be permitted for both ON_ENTRY and ON_EXIT.

# 3.3.6.2 34 A

I would suggest that instead of reconstructing the source, that the actual source be accessed via pointers that exist in the either the retained AST, DIANA or the statement table.

I question whether the pretty printer is going to be able to pick up formatting in the middle of a program at any arbitrary line number.

^L

!SPECIFICATION TITLE: VIRTUAL MEMORY METHODOLOGY
!SPECIFICATION DATE: 82 OCT 8
!SPECIFICATION NUMBER: IR-MA-142-1
!REVIEW DATE: 83 MAY 1

Paragraph	Page	Competence Level
# 0.0	GENERAL	B

None of the B5 specifications examined contain an index. This proved extremely bothersome. In many instances I wished to compare all uses of a particular term, find the definition of a term, and the like. Without an index such tasks are effectively impossible. This is quite simply unacceptable.

GENERAL B

I was most disappointed to get to the detailed information on the various facilities where the package specifications are given, or at least sketched, and find only two or three comment lines per procedure or

function that can be invoked by the user. A completely formal specification is, admittedly, neither desirable nor tractable. However, a more robust specification technique could have been adopted to describe WHAT the operation does; the interface alone is just too little information. In general, this lack of information prevented me from throughly comparing the package specification with the other written description of functionality that was provided. The information I needed to make such a comparison was simply missing - and I did not feel like "guessing", and then critiquing my own guess.

0 A

This package looks very capable but I am concerned that the use of a virtual database by either the co e tcomplex. I can't point tolar features as excess, but the structure seems ready to fall of its own weight. Compromises introduced early in the design seem to have caused other features to be added which in turn required the original compromise to be necessary.

Take the Rep Analyzer for example. It is required because the VMM system places severe restrictions on the components of records in the VMM system. But most of these restr:ctions are there to make the Rep Analyzer easy to implement, and to implement the concept of a domain. But domains are only needed to allow mixing of VMM designator types in records, which can not otherwise be done given the existence of the Rep Analyzer, and so on. (Why can't I have an access value in a VMM object? It doesn't have to designate a VMM object, it could designate something else. For example, if the compiler maintained a symbol table in memory but the Diana tree was in VMM, then a tree node could designate a symbol table entry larger than the data it indexes.)

To sum up I think that a searching design review could come up with a design which provides more functionality and is much easier to implement. We have done it here, and although the goals and assumptions about the underlying hardware are different, we had no trouble staying inside Ada and implementing a virtual memory capability which places very few restrictions on the user. (And if the the "user" is the compiler's code generator, the end user need know nothing about the underlying system to use it.)

# 3.1	5	B

Throughout the specification, it is somewhat unclear whether the facilities offered by the VMM are in fact complete enough to be usable for purposes other than implementing the CPCI COMP.DIANA. Put another way, this facility is necessary to implement certain dependent portions of the AIE; how useful/complete will it be to the user of the AIE?

# 3.2.4.2.1	7	B

The manner in which operations are associated with the type specified in the user supplied package specification is unclear. It is also unclear what constraints, if any, are applicable to the enumeration type that is used as the for the variant record. It later (3.2.4.2.2) becomes somewhat more clear that the operations on the data type instances are those provided by the VMM primitive operations.

	9	B

The block immediate beneath the "MAPSE Tool" block in figure 3-1 is unlabeled. In as much as it is shown to be output of the Representation Analyzer, a label is deemed critical.

# 3.3.1.1	11	B

The architecture of the Rep Analyzer is such that all types which any given tool wishes to access with the VMM facility must be processed as a single collection (Figure 3-2, pg 10). The obvious consequence of this is that a change in any of the types forces a regeneration of the VMM_access_package source, and hence recompilation of the using packages. It is questioned whether the rep analyzer could not have been designed so as to allow multiple VMM_access_packages. The goal of this alternative being the minimization of recompilation. If this option is available, it is not clearly stated as an option.

12 B

The exact form the of the pragma that must be included in user supplied package specification input to the Rep Analyzer is unspecified. Likewise for the the WITH clause used to access VMM generic packages. It is unclear whether these requirements are a reasonable imposition on the user of the Rep Analyzer, or could become unnecessarily burdensome.

# 3.3.1.2 13 B

The phrase "Verify restrictions and conventions" is needlessly vague. If a later paragraph explains this processing further, a forward reference to the paragraph or paragraphs should be included.

# 3.3.2.1 15 B

The meaning of the sentence "It will also be possible to specify a complete implementation for virtual records components by writing procedure bodies for the attribute selectors." is unclear.

# 3.3.2.2 16 B

The built-in encapsulated types are, in general, underspecified. Enough information is not provided to allow determination of the exact facilities that will be provided, and their characteristics.

# 3.3.2.2.1 18 B

No mention is made of how to create new elements of the dynamic vector, or how to delete existing elements. As described, the vector is of static size, rather than dynamic. In addition, the type of the indices is unspecified. In general the description is too sketchy to convey complete understanding of what is to be provided.

# 3.3.3.1 20 B

It is unclear whether the mode is permanently associated with the created VMSD or not. If not, the mode is probably effective only for (a portion of) the invocation of the program creating the VMSD.

# 3.3.3.3 ,4 23-29 B

Good discussion of the alternative designs, and their relative merits.

# 3.3.3.6 32 B

A 32 bit integer may not prove an adequate implementation for page buffer sequence numbers. In particular, it would seem a poor choice for application programs utilizing the VMM facility that are intended to run perpetually.

ADA INTEGRATED ENVIRONMENT

PHASE II EVALUATION CRITERIA

The questions below have been answered with a ranking of 1 to 5 with 1 being least satisfactory; in some instances the answer has been amplified with a comment.

I. GENERAL

A. To what extent does the design conform to the KAPSE/MAPSE approach for an integrated programming environment? 5

To what extent does the design satisfy the design goals stipulated as general guidelines in STONEMAN? 5

C. Does the overall design provide a system that is conducive to the development and maintenance of quality Ada software? 3 The debugging facilities are weak in some practical machine level features.

D. Does the system design facilitate the development and integration of new tools? 5

E. Does it provide for improvements, upgrades and modifications to be accomplished in an easy manner? ?

F. Has the contractor given consideration to and provided for the addition of tools to support development throughout the software life cycle? 3 The embedded computer support is weak but unfortunately according to the SOW.

G. To what degree does the system support both programming and project management functions? 4

H. Has the system been human engineered to provide a consistent user interface that is easy to learn and use? 3 Some of the commands are too verbose.

I. Does the system provide basic facilities to establish project and configuration management controls? 5

J. Does the system facilitate portability? 4 According to the specification; however, it would be nice if the contract would require proof by a second, different host.

K. Are all host dependent interfaces clearly identified and specified? 5

L. Are the host dependent interfaces minimized and isolated to the maximum extent possible? 5

M. What is the degree of effort required to rehost the system? ?

N. Has the system been designed to exploit, but not demand, modern high capacity and high performance host system software? 1 The VMM facility will probably sap resources. The system will likely not fit on small computers which is unfortunate.

O. Are communication protocols and conventions uniform and consistent throughout the system? 4

P. Does the system provide self-protection and self-recovery facilities as well as provide user and project protection? 5

Q. Does the design provide for a truly integrated system? 5

R. Does the design provide for software reusability wherever possible and reasonable? 5

S. To what extent does the design achieve the goal of granularity as stated in STONEMAN? 5

T. Does the design provide for sharing of data and functions? 4

U. Does the overall approach reflect a detailed analysis of all system requirements? 4

V. Has the contractor performed reasonable trade-off analyses and selected the optimum approach? 3

W. Are the deviations or additions (if any) to the system proposed by the contractor beneficial to the government? ?

X. As a result of its analysis and design, has the contractor identified or provided additional capabilities that can be implemented without additional cost?

Y. To what extent does the design support the model of an APSE running on a host machine and supporting development of software for an embedded target machine? 1 This appears to be the least addressed capability.

II. KAPSE DATABASE

A. Does the database system design provide complete database facilities to support the development, management and maintenance of Ada programs? 4

B. Does the database system design provide comprehensive database operations and access facilities to meet user, project manager and MAPSE tool needs? 4

D. Does the design provide an integrated and flexible approach to versions, attributes, partitions and access controls? 5

III. INTERFACES

A. Does the contractor have a sound approach for realizing the concept of a virtual interface? 4

B. To what extent do the database facilities specified support the development, management and maintenance of Ada programs? 4

C. Are all the interfaces specified in the design consistent, straightforward and do they facilitate user and tool communication? 5

IV. KAPSE FUNCTIONS

A. Does the design meet the requirements for the KAPSE functions specified in the S.O.W.? 5

V. MAPSE A. Do the designs for the MAPSE tools satisfy requirements for portability, modularity, flexibility and ease of use? 4

B. Are user/tool communication conventions designed to allow the user to communicate in a natural and consistent manner across all tools? 4

C. Does the contractor indicate an intention to develop all MAPSE tools in Ada? 5 If not, are all exceptions justified?

D. Do the MAPSE tool designs meet or exceed the requirements established in the S.O.W.? 5

VI. EDITOR

[This section of the Air Force criteria is irrelevant since the contract for ALS does not currently require an editor and for the time being the Army intends to use the DEC VAX/VMS supplied editor.\\However, for completeness, and to allow any comments reviewers wish to make with respect to editor capabilities,...]

A. Do the contractor's specifications provide a sound approach for furnishing the capabilities required in a basic editing facility? No specification was received for the AIE Editor.

VII. DEBUGGER

A. Does the debugging facility design thoroughly address batch and on-line users in detecting, locating and correcting errors in Ada programs? 3 The debugger commands are too verbose and do not address machine level debugging.

VIII. COMPILER

A. Is the Ada compiler design modular and within the designated system resources? The stated objectives of the compiler specification is to operate within 512KB but to utilize more memory when available to improve compiler performance. I believe this conforms to the SOW requirements but that size as a minimum configuration will limit the potential hosts for the AIE and that seems unfortunate.

B. Does the design present a state-of-the-art approach to rehosting and retargeting the Ada compiler without undue difficulty? Although it is not at all clear how difficult rehosting and retargeting will be, the VMM schema should contribute to the rehostability of the compiler. It may be that the emphasis on rehostability and the use of VMM may seriously impact compiler performance.

C. Does the Ada compiler design satisfy the input/output requirements stipulated in the S.O.W.? 4

D. Is the contractor's approach to error analysis extensive? The mechanisms appear to be there but the proof in this area is in the implementation.

E. Is the optimizer design state-of-the-art? Do the techniques selected justify the time/resources trade-off? The spec btr fon. Both should be done.

F. Does the contractor propose to meet the compiler performance and capacity specifications of the S.O.W.? It appears that one or the other is permitted but not both.

IX. LINK/LOADER

A. Will the design of the linking and loading tools provide a smooth flow to executing Ada programs? The linker tool completes the Ada program construction requirement and complies with the Ada requirements of program component compatibility. Although there appears to be some user control over image storage allocation; there seems no description of how to load or invoke such overlays.

X. Ada PROGRAM LIBRARY

A. Is the design of the Ada program library optimum?

The AIE program library facilities appear very comprehensive although it is difficult to understand some of the documentation regarding it. My concern would be that a substantial overhead will accrue from its management.

XI. PROJECT/CONFIGURATION MANAGEMENT FACILITIES

A. In support of project/configuration management, does the design provide reports for batch and on-line users, some reports and automatic stub generation, as a minimum? The distinction between batch and on-line users is unclear; at times it appears that the commands are identical but the interaction with the input and output devices/objects may be inconsistent. For purposes of providing reports and for command language preparation, the facilities are more than sufficient. Insufficient detail is given to determine if the output of the CM tool, i.e., PIF, is adequate. PIF does provide for automatic stub generation.

XII. HIGH LEVEL I/O

A. Is the I/O package design a valid extension or alternate to the Ada Language Manual? 5

XIII. TERMINAL INTERFACE ROUTINES

A. Do the design terminal interface routines proposed satisfy the requirement? Not enough detail to determine yet.

XIV. QUALITY ASSURANCE, TESTING, VALIDATION

A. As a minimum, does the proposed computer program development, quality assurance, testing and validation plan satisfy the requirements in the S.O.W.? 4